

Redis 设计与实现

黄健宏 著

The Design and Implementation of Redis

- 系统而全面地描述了 Redis 内部运行机制。
- 图示丰富，描述清晰，并给出大量参考信息，是NoSQL数据库开发人员案头必备。
- 包括大部分Redis单机特征，以及所有多机特性。



机械工业出版社
China Machine Press

□ □ □ □ □ □ □

Redis

100

ISBN 978-7-111-46474-7

□□□□□□□□□□2014□□□□□□□□□□□□□□□□□□□□□□
□□□□

□ □ □ □ □ □ □ □

☐☐☐☐☐+ 86-10-68995265

□□□□service@bbbvip.com

□□□□□www.hzmedia.com.cn

□□□□ @□□□□

📧 @yanfabook

目次

目次

目次

第1章 目次

1.1 Redisのインストール

1.2 目次

1.3 目次

1.4 目次

1.5 目次

第2章 目次

2.1 目次

2.1 SDSのインストール

2.2 SDSのCのインストール

2.3 SDS API

2.4 目次

2.5 目次

第3章 目次

3.1 目次

3.2 目次API

3.3 目次

第4章 索引

4.1 索引类型

4.2 索引的创建

4.3 索引的删除

4.4 rehash

4.5 索引的rehash

4.6 索引API

4.7 索引的维护

第5章 视图

5.1 视图的创建

5.2 视图的API

5.3 视图的维护

第6章 触发器

6.1 触发器的创建

6.2 触发器的删除

6.3 触发器的维护

6.4 触发器的API

6.5 触发器的API

6.6 触发器的维护

第7章 存储过程

7.1 存储过程的创建

7.2 環境変数

7.3 環境

7.4 環境API

7.5 環境

8 環境

8.1 環境変数

8.2 環境

8.3 環境

8.4 環境

8.5 環境

8.6 環境

8.7 環境変数

8.8 環境

8.9 環境

8.10 環境変数

8.11 環境

環境変数

9 環境

9.1 環境変数

9.2 環境

9.3 環境

9.4 配置持久化策略

9.5 配置快照

9.6 Redis持久化策略

9.7 AOF与RDB持久化策略

9.8 配置策略

9.9 配置策略

10 RDB策略

10.1 RDB策略配置

10.2 配置策略

10.3 RDB策略

10.4 RDB策略

10.5 配置策略

10.6 配置策略

11 AOF策略

11.1 AOF策略配置

11.2 AOF策略配置

11.3 AOF策略

11.4 配置策略

12 策略

12.1 配置策略

12.2 配置策略

12.3 環境構築

12.4 実行

12.5 確認

13 実行

13.1 実行

13.2 実行

13.3 確認

14 実行

14.1 実行

14.2 serverCron実行

14.3 実行

14.4 確認

15 実行

15 実行

15.1 実行

15.2 実行

15.3 実行

15.4 実行

15.5 PSYNC実行

15.6 実行

15.7 確認

15.8 哨兵

16 Sentinel

16.1 哨兵 Sentinel

16.2 哨兵 Sentinel

16.3 哨兵 Sentinel

16.4 哨兵 Sentinel

16.5 哨兵 Sentinel

16.6 哨兵 Sentinel

16.7 哨兵 Sentinel

16.8 哨兵 Sentinel

16.9 哨兵

16.10 哨兵

16.11 哨兵

17 哨兵

17.1 哨兵

17.2 哨兵

17.3 哨兵 Sentinel

17.4 哨兵

17.5 ASK 哨兵

17.6 哨兵 Sentinel

17.7 哨兵

17.8 17.8

17.8

18 18

18.1 18.1

18.2 18.2

18.3 18.3

18.4 18.4

18.5 18.5

18.6 18.6

19 19

19.1 19.1

19.2 WATCH 19.2

19.3 ACID 19.3

19.4 19.4

19.5 19.5

20 Lua 20

20.1 Lua 20.1

20.2 Lua 20.2

20.3 EVAL 20.3

20.4 EVALSHA 20.4

20.5 20.5

20.6 関数

20.7 関数

20.8 関数

第21章 関数

21.1 SORT関数

21.2 ALPHA関数

21.3 ASC関数DESC関数

21.4 BY関数

21.5 関数ALPHA関数BY関数

21.6 LIMIT関数

21.7 GET関数

21.8 STORE関数

21.9 関数関数

21.10 関数

第22章 関数関数

22.1 関数関数

22.2 GETBIT関数

22.3 SETBIT関数

22.4 BITCOUNT関数

22.5 BITOP関数

22.6 関数

22.7 閉空間

23 閉空間

23.1 閉空間の定義

23.2 閉空間の性質

23.3 閉空間の例

23.4 閉空間の応用

24 閉空間

24.1 閉空間の定義

24.2 閉空間の性質

24.3 閉空間の例



□□□□2011□4□□□□□□□□□□□□□□□□□□□□□□□□“□□□□”□□□
□□□□□□□□□□□□□□□□

```

    00000000huangz0000peter0tom0jack0000000john0000peter0
tom0bob0david0000000000huangz00john00000000000000000000
0000“00john00000peter0tom”00000000

```

[illegible]

Redis Redis—— Redis

Redis

Redis

Redis

SQL

Redis

Redis Redis
Redis

·Redis

·Redis “hello world”
10086 3.14 SETBIT Redis

·Redis APPEND HSET
DEL TYPE EXPIRE
Redis

·Redis

·Redis

·Redis

Redis
Redis Redis Redis

Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など

Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など
2012年11月11日
2013年3月8日

Redisのバージョンは2.0、2.2、2.4、2.6など

Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など

Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など

Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など
Redisのバージョンは2.0、2.2、2.4、2.6など

2013年3月2014年1月11日
Redis unstable
Redis 3.0
Redis

```

bitop Sentinel
Redis Redis
Redis Lua
Redis Sentinel

```

Redis
Redis
Redis
Redis
Redis

www.RedisBook.com
[disqus](https://disqus.com)

Redis

Redis

□□□□huangz□

2014年3月



hoterran
iammutex

Redis

TimYang

Redis Salvatore Sanfilippo

Redis

第1章 简介

Redis 是一个开源的、基于内存的、高性能的键值数据库。它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis 的特点包括：快速、简单、持久化、分布式、集群化。

Redis 的架构非常简单，它由一个主进程和多个从进程组成。主进程负责处理客户端的请求，从进程负责复制主进程的数据。Redis 还支持主从复制、哨兵、集群等多种部署方式。

Redis 的持久化机制非常灵活，它支持两种持久化方式：RDB 和 AOF。RDB 是将内存中的数据快照保存到磁盘，AOF 则是将客户端的每一个写操作都记录到日志中。Redis 还支持混合持久化，即同时使用 RDB 和 AOF。

Redis 的社区非常活跃，它拥有大量的用户和开发者。Redis 的官方网站是 [RedisBook.com](http://redis.io)，它提供了 Redis 的官方文档、教程、社区资源等。Redis 的源代码是公开的，你可以在 GitHub 上找到 Redis 的源代码。

1.1 Redis 的安装

目前 Redis 2.9——Redis 3.0 的安装方法基本一致。Redis 3.0 的安装方法与 Redis 2.6 和 Redis 2.8 的安装方法基本一致。Redis 2.6 和 Redis 3.0 的安装方法基本一致。

Redis 的安装方法基本一致。Redis 3.0 的安装方法基本一致。

1.2 Redis

Redis 是一个开源的、基于内存的、使用 C 语言编写的、支持多种数据结构的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis 还支持持久化、主从复制、事务、Lua 脚本等功能。

Redis 是一个开源的、基于内存的、使用 C 语言编写的、支持多种数据结构的数据库。

Redis 是一个开源的、基于内存的、使用 C 语言编写的、支持多种数据结构的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis 还支持持久化、主从复制、事务、Lua 脚本等功能。

- 字符串对象 (string object)
- 列表对象 (list object)
- 集合对象 (set object)
- 有序集合对象 (sorted set object)
- 哈希表对象 (hash object)

Redis 是一个开源的、基于内存的、使用 C 语言编写的、支持多种数据结构的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis 还支持持久化、主从复制、事务、Lua 脚本等功能。

```
redis> SET msg "hello world"
OK
```

Redis 是一个开源的、基于内存的、使用 C 语言编写的、支持多种数据结构的数据库。它支持字符串、列表、集合、有序集合、哈希表、位图等数据类型。Redis 还支持持久化、主从复制、事务、Lua 脚本等功能。

```
redis> RPUSH numbers 1 3 5 7 9
(integer) 5
```

Redis 的持久化功能，可以将内存中的数据持久化到磁盘上，防止数据丢失。Redis 提供了两种持久化方式：RDB 和 AOF。

Redis 持久化方式

Redis 提供了两种持久化方式：

1. RDB (Redis Database Backup)：将内存中的数据快照保存到磁盘上。RDB 文件是一个二进制文件，可以通过 `redis-cli` 命令进行备份和恢复。

2. AOF (Append Only File)：将 Redis 接收到的每个写操作命令按顺序记录到磁盘上的一个文件中。AOF 文件是一个文本文件，可以通过 `redis-cli` 命令进行备份和恢复。AOF 文件支持 `BGSAVE` 和 `BGREWRITEAOF` 命令。

Redis 提供了两种持久化方式：

1. `accept`：接受客户端的连接请求。Redis 的 `accept` 函数用于接受客户端的连接请求，并返回一个套接字。

2. `redis.c/serverCron`：Redis 的 `serverCron` 函数用于处理 Redis 的持久化操作。Redis 的 `serverCron` 函数会定期检查 Redis 的持久化状态，并根据需要进行持久化操作。

Redis 提供了两种持久化方式：

1. RDB (Redis Database Backup)：将内存中的数据快照保存到磁盘上。RDB 文件是一个二进制文件，可以通过 `redis-cli` 命令进行备份和恢复。

2. AOF (Append Only File)：将 Redis 接收到的每个写操作命令按顺序记录到磁盘上的一个文件中。AOF 文件是一个文本文件，可以通过 `redis-cli` 命令进行备份和恢复。AOF 文件支持 `BGSAVE` 和 `BGREWRITEAOF` 命令。

14 “ ” Redis serverCron Redis

“ ”

Redis Sentinel replication cluster

15 “ ” Redis master-slave replication

16 “Sentinel” Redis Sentinel Sentinel Sentinel

17 “ ” Redis node redirection

“ ”

Redis

18 “ ” PUBLISH SUBSCRIBE PUBSUB Redis

19“”MULTIEXECWATCHRedis
RedisACID

```

20 "Lua" EVAL EVALSHA SCRIPT LOAD
Redis Lua Redis
Lua Lua

```

```
021" "SORTSORTDESCALPHA
GETSORT
```

22 “” Redis GETBIT SETBIT BITCOUNT BITOP

23 “Redis”slow log

24 “ ” monitor

1.3 Redis 部署

Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。

Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。

Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。

Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。Redis 部署方案主要分为单机部署和集群部署。单机部署适用于小规模应用，集群部署适用于大规模应用。

1-1 Redis 部署方案

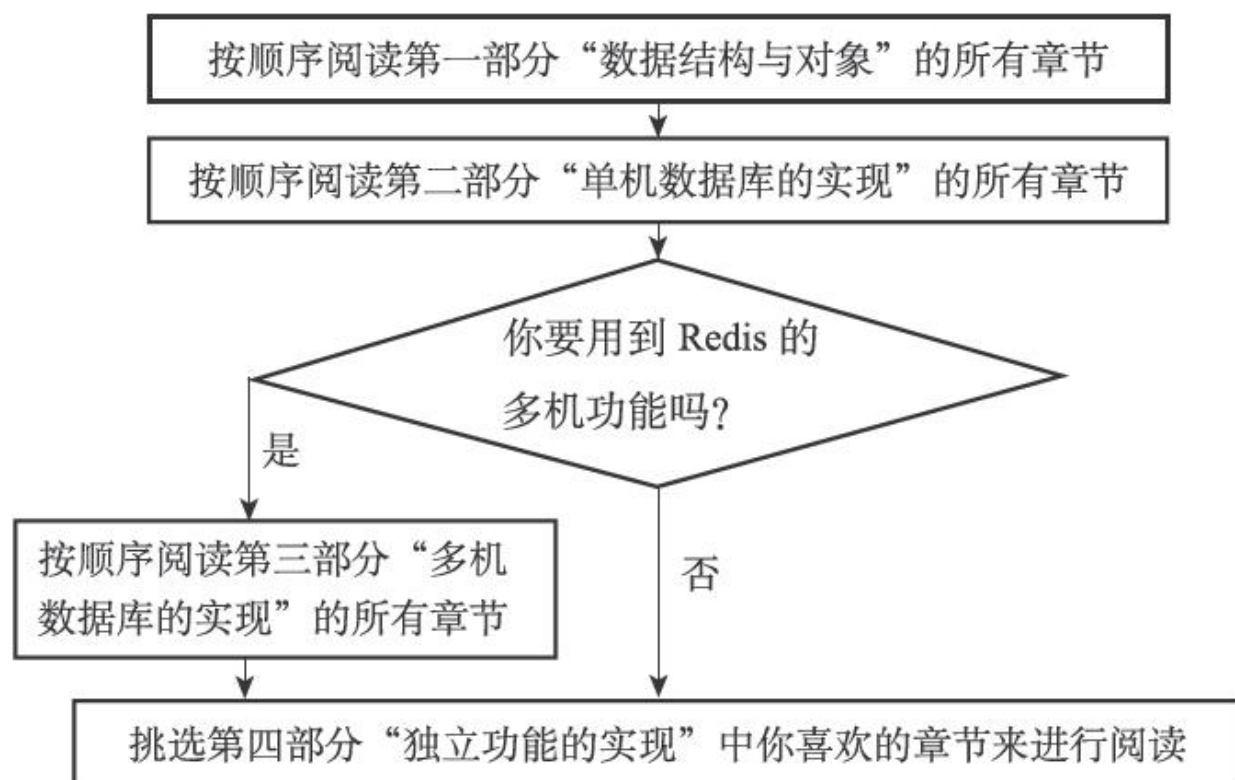


图1-1 学习路径

1.4 〇〇〇〇

--	--	--	--	--	--

```

Redisfile namefile/name
redis.c/mainredis.cmainredis.h/redisDb
redis.hredisDb

```

```

file name <file>/name
<unistd.h>/write unistd.h write <stdio.h>/printf
stdio.h printf

```

```
redis.h redisDb redis.h/redisDb
redisDb redisDb
```

--	--	--	--	--	--

```
struct.property struct property
redisDb.id redisDb id redisDb.expires redisDb
expires
```

redis 的 安装 与 使用

redis

redis 的 C 语言 Python 接口

- redis 的 C 语言接口 redis 的 C 语言接口

- redis 的 Python 接口 redis 的 Python 接口

redis 的 Python 接口 server client 接口

redis.h/redisServer 接口 client 接口

redis.h/redisClient 接口

1.5 安装

访问redisbook.com获取Redis的安装指南。Redis的安装指南位于redisbook.com/redis/install。

□□□□ □□□□□□□□

□2□ □□□□□□□

□3□ □□

□4□ □□

□5□ □□□

□6□ □□□□

□7□ □□□□

□8□ □□

2 字符串

Redis 使用 C 语言实现的简单动态字符串（simple dynamic string，SDS）来存储字符串。SDS 是 Redis 内部使用的一种

Redis 使用 C 语言实现的 string literal 来存储字符串。

```
redisLog(REDIS_WARNING,"Redis is now ready to exit, bye bye...");
```

Redis 使用 SDS 来存储字符串。Redis 使用 SDS 来存储字符串。

```
redis> SET msg "hello world"
OK
```

Redis 使用 SDS 来存储字符串。

· Redis 使用 SDS 来存储字符串。

· 문자열을 저장하는 Redis의 기본 데이터 구조는 SDS입니다. “hello world”

SDS

Redis는 SDS를 사용하여 문자열을 저장합니다.

```
redis> RPush fruits "apple" "banana" "cherry"
(integer) 3
```

Redis는 Redis를 사용하여 문자열을 저장합니다.

· Redis는 Redis를 사용하여 문자열을 저장합니다. “fruits” SDS

· Redis는 Redis를 사용하여 문자열을 저장합니다. SDS

Redis는 Redis를 사용하여 문자열을 저장합니다. “apple” SDS “banana” SDS “cherry”

Redis는 Redis를 사용하여 문자열을 저장합니다. SDS buffer AOF AOF
Redis는 Redis를 사용하여 문자열을 저장합니다. SDS AOF
Redis는 Redis를 사용하여 문자열을 저장합니다. SDS

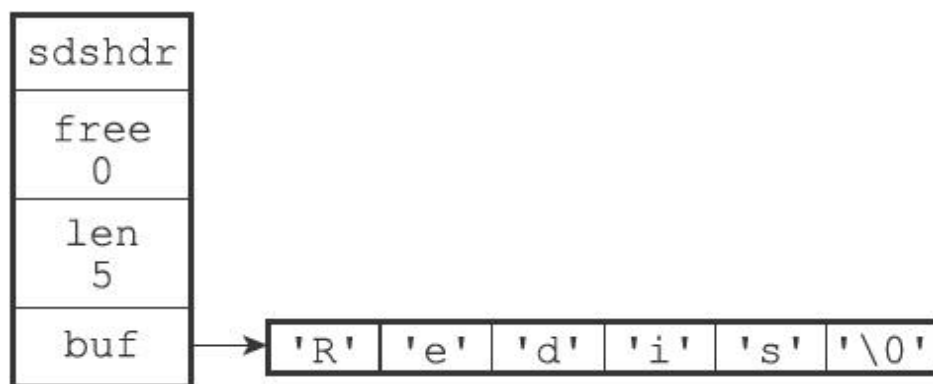
Redis는 Redis를 사용하여 문자열을 저장합니다. SDS Redis
SDS C API

2.1 SDS

sds.h/sdshdr SDS

```
struct sdshdr {  
    //  
    buf  
    //  
    SDS  
    int len;  
    //  
    buf  
    int free;  
    //  
    char buf[];  
};
```

2-1 SDS



2-1 SDS

·free 0 SDS

·len 5 SDS

·buf char 'R' 'e' 'd' 'i' 's' '\0'

SDS C 1 SDS len
1 SDS
SDS C

2-1 SDS s
<stdio.h>/printf

```
printf("%s", s->buf);
```

SDS "Redis" SDS

2-2 SDS SDS SDS
"Redis" SDS SDS buf
free 5

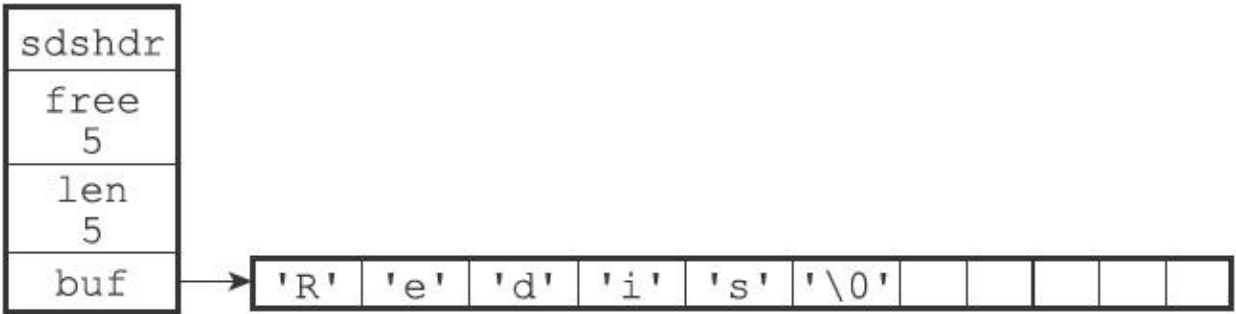


图2-2 SDS的内存结构

SDS字符串的内存结构

2.2 SDS C 实现

在 C 语言中，字符串是以字符数组的形式存储的。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。

图 2-3 展示了 Redis 中字符串的存储方式。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。

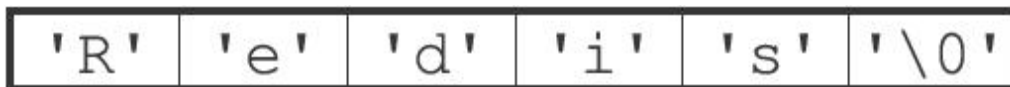


图 2-3 C 实现

在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。

2.2.1 字符串的存储

在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。

图 2-4 展示了 Redis 中字符串的存储方式。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。在 Redis 中，字符串是以 SDS 的形式存储的。SDS 是 Simple Dynamic String 的缩写。

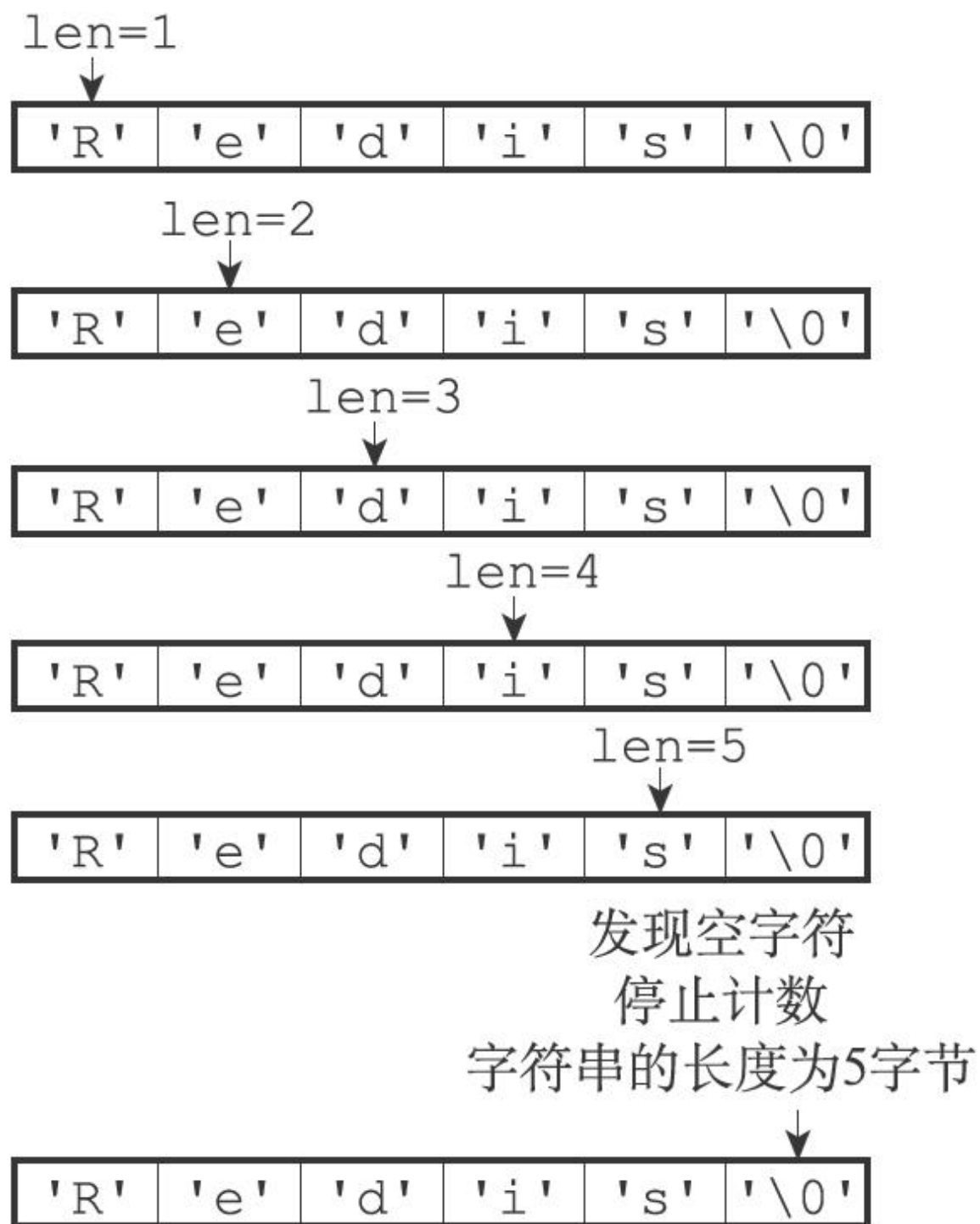


图2-4 字符串长度的计算

图C展示了SDS len 0和1的情况

图2-5展示了SDS len 5的情况

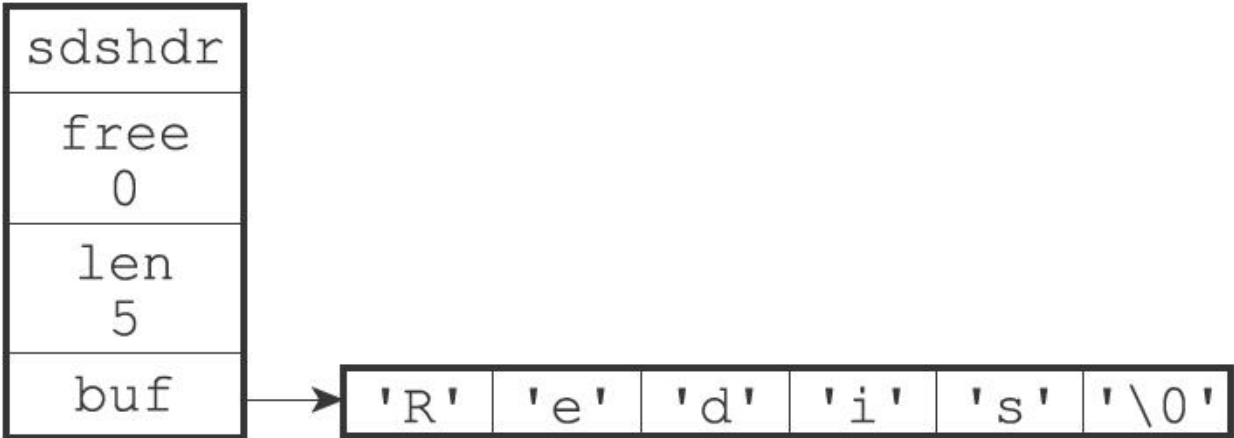


图2-5 5字节的SDS

图2-6展示了SDS len 11的情况

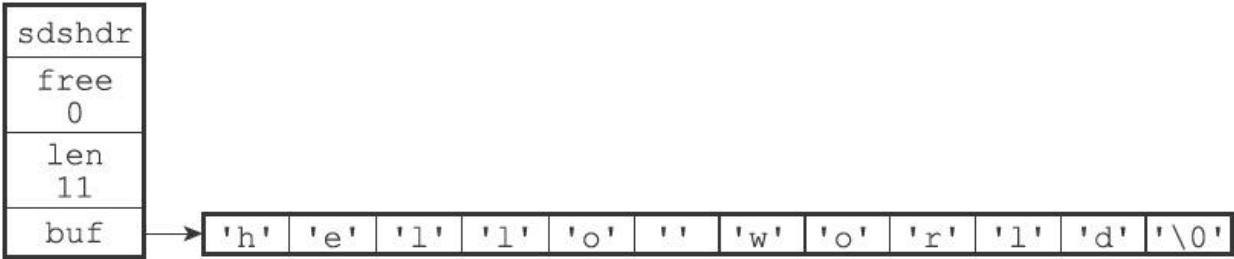


图2-6 11字节的SDS

Redis SDS 是 Redis 的字符串 API，它提供了对字符串的增删改查操作。

Redis SDS 是 C 语言实现的，它使用 O(1) 的时间复杂度来操作字符串。Redis 使用 SDS 来存储字符串，它使用 STRLEN 来操作字符串，STRLEN 的时间复杂度是 O(1)。

2.2.2 字符串操作

在 C 语言中，字符串操作通常使用 `strcat` 函数，它会将 `src` 字符串追加到 `dest` 字符串的末尾。

```
char *strcat(char *dest, const char *src);
```

C 语言中的 `strcat` 函数会将 `src` 字符串追加到 `dest` 字符串的末尾。

在 Redis 中，我们使用 `s1` 和 `s2` 两个字符串，`s1` 是 "Redis"，`s2` 是 "MongoDB"。图 2-7 展示了字符串操作。

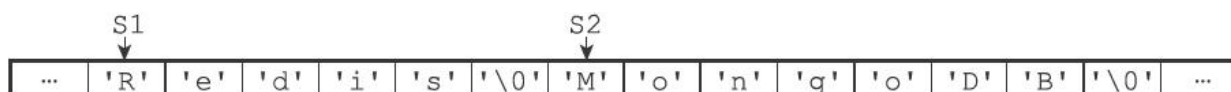


图 2-7 字符串操作

Redis Cluster

```
strcat(s1, " Cluster");
```

s1 指向 "Redis Cluster" 字符串的末尾，strcat 从 s1 指向的位置开始，将 s2 指向的字符串复制到 s1 指向的位置。图 2-8 所示。

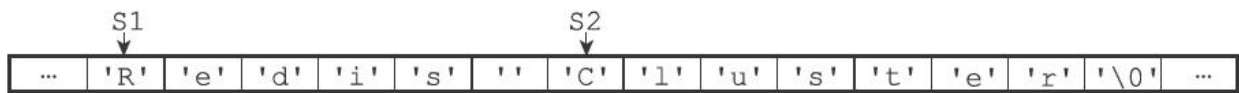


图 2-8 s1 指向 s2 指向的位置

C 语言提供了 SDS 库，用于管理字符串。SDS API 提供了 SDS 库的 API，用于管理字符串。SDS 库提供了 API，用于管理字符串。SDS 库提供了 API，用于管理字符串。SDS 库提供了 API，用于管理字符串。

SDS API 提供了 sdscat 函数，用于将 C 语言字符串复制到 SDS 库中。sdscat 函数将 C 语言字符串复制到 SDS 库中。sdscat 函数将 C 语言字符串复制到 SDS 库中。

Redis Cluster

```
sdscat(s, " Cluster");
```

在 SDS 的 s 2-9 中，sdscat 函数将字符串 s 复制到 buf 中，并返回指向 buf 的指针。在图 2-9 中，sdscat 函数将字符串 "Redis" 复制到 buf 中，并返回指向 buf 的指针。在图 2-10 中，sdscat 函数将字符串 "RedisCluster" 复制到 buf 中，并返回指向 buf 的指针。



图 2-9 sdscat 函数与 SDS



图 2-10 sdscat 函数与 SDS

在图 2-10 中，SDS 的 sdscat 函数将字符串 "RedisCluster" 复制到 buf 中，并返回指向 buf 的指针。在图 2-10 中，SDS 的 sdscat 函数将字符串 "RedisCluster" 复制到 buf 中，并返回指向 buf 的指针。在图 2-10 中，SDS 的 sdscat 函数将字符串 "RedisCluster" 复制到 buf 中，并返回指向 buf 的指针。

2.2.3 字符串的内存管理

在 Redis 中，字符串的内存管理是通过 SDS 来实现的。在 Redis 中，字符串的内存管理是通过 SDS 来实现的。在 Redis 中，字符串的内存管理是通过 SDS 来实现的。

sdscat(s, "Tutorial");

sdscat 13 9
"Tutorial" SDS 2-13



2-13 sdscat SDS

SDS API SDS API

SDS N N N

2.

SDS SDS API SDS free

sdstrim SDS C SDS C

图2-14 SDS s

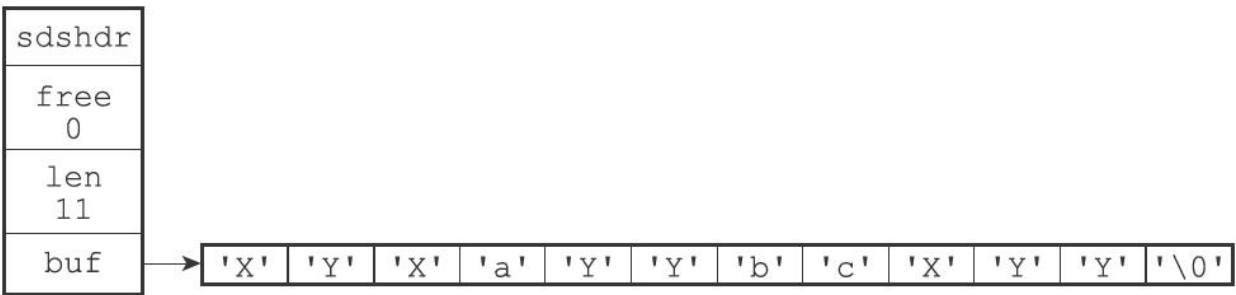


图2-14 SDS s

```
sdstrim(s, "XY"); //
SDS
'X'
'Y'
```

图2-15 SDS

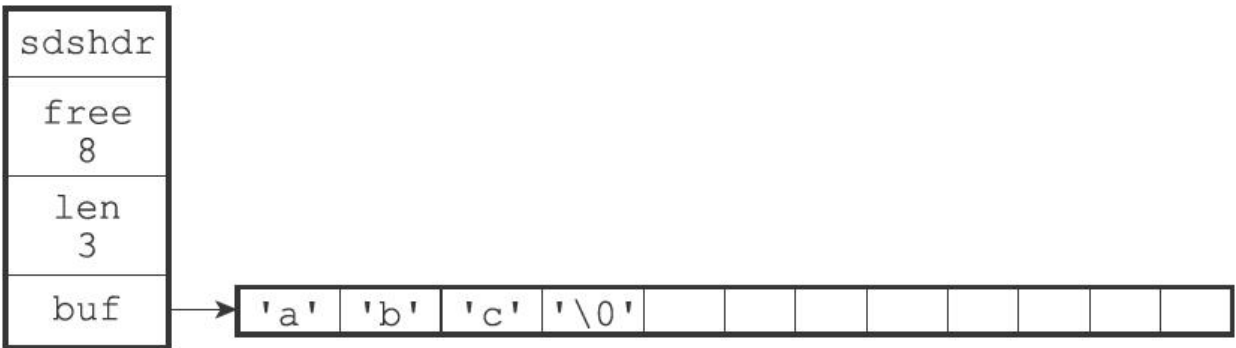


图2-15 SDS

sdstrim SDS 8 8

SDS SDS

s


```
sdscat(s, " Redis");
```

sdscat 函数调用 SDS 函数 8 次，每次调用 6 个字节，"Redis" 字符串 2-16 字节。

sdscat 函数调用 SDS 函数 8 次，每次调用 6 个字节，"Redis" 字符串 2-16 字节。

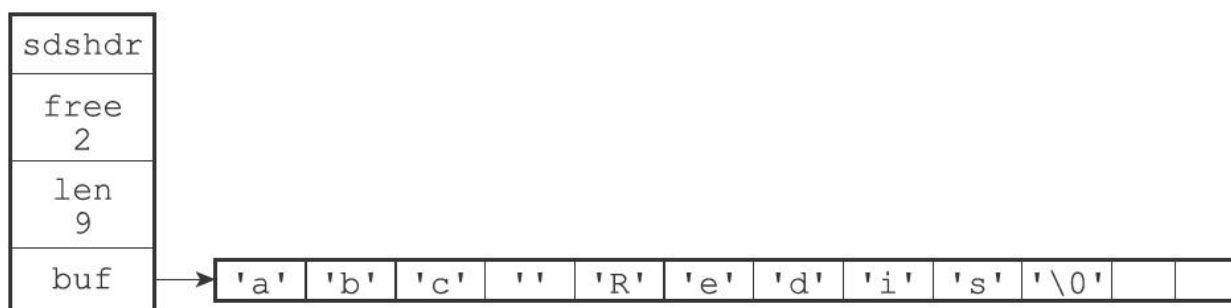


图 2-16 sdscat 函数 SDS

SDS 函数 API 调用 SDS 函数 8 次，每次调用 6 个字节，"Redis" 字符串 2-16 字节。

2.2.4 字符串

C 语言字符串使用 ASCII 码表示，每个字符占用 1 个字节。C 语言字符串使用 ASCII 码表示，每个字符占用 1 个字节。

图 2-17 展示了 C 语言字符串 "Redis" 和 "Cluster" 的内存布局。



图2-17 字符串的内存布局

Redis 使用 SDS 来存储字符串。SDS API 是 binary-safe 的。SDS API 使用 buf 来存储字符串。SDS 使用 len 来存储字符串的长度。SDS 使用 free 来存储字符串的起始地址。SDS 使用 len 来存储字符串的长度。SDS 使用 free 来存储字符串的起始地址。

Redis 使用 SDS 来存储字符串。SDS API 是 binary-safe 的。SDS API 使用 buf 来存储字符串。SDS 使用 len 来存储字符串的长度。SDS 使用 free 来存储字符串的起始地址。

Redis 使用 SDS 来存储字符串。SDS API 是 binary-safe 的。SDS API 使用 buf 来存储字符串。SDS 使用 len 来存储字符串的长度。SDS 使用 free 来存储字符串的起始地址。

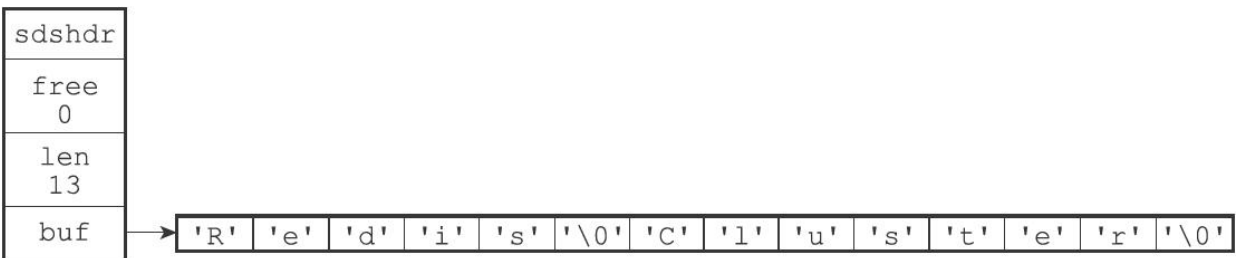


图2-18 SDS 的内存布局

Redis 使用 SDS 来存储字符串。SDS API 是 binary-safe 的。SDS API 使用 buf 来存储字符串。SDS 使用 len 来存储字符串的长度。SDS 使用 free 来存储字符串的起始地址。

2.2.5 C 语言

SDS API 提供了非常简单的 C 语言字符串 API。
 SDS 字符串结构体包含一个 buf 指针，指向字符串的起始位置。
 使用 SDS 字符串结构体，需要包含 <string.h> 头文件。

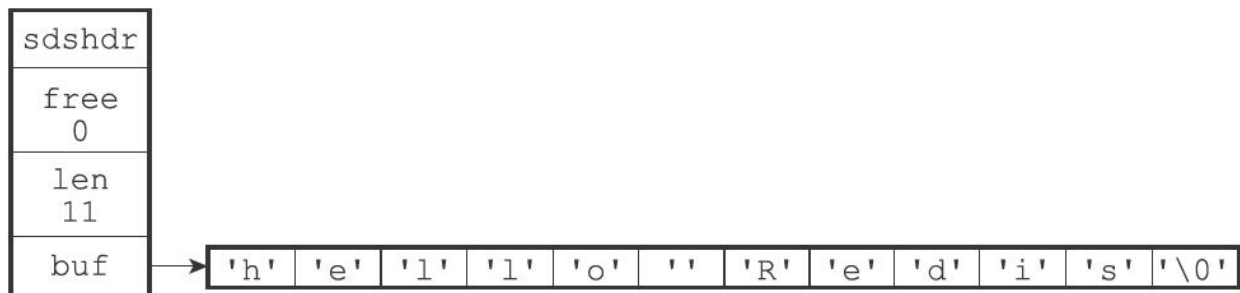


图 2-19 SDS 字符串结构体

如图 2-19 所示，SDS 字符串结构体 sds 包含一个指向字符串的指针 buf。
 使用 <string.h>/strcasecmp 函数可以比较两个 SDS 字符串。

```
strcasecmp(sds->buf, "hello world");
```

Redis 使用 SDS 字符串结构体来存储字符串。

使用 SDS 字符串结构体，可以使用 strcat 函数将两个 SDS 字符串连接起来。
 使用 C 语言字符串结构体，可以使用 strcat 函数。

```
strcat(c_string, sds->buf);
```

Redis 使用 SDS 字符串结构体来存储字符串。

在C语言中，字符串是以字符数组的形式存储的。而SDS（Simple Dynamic String）是Redis中使用的字符串类型，它是在C语言的基础上，通过引入动态字符串的概念，对字符串进行了封装。SDS的定义在<string.h>头文件中。

2.2.6 SDS

图2-1 C语言与SDS字符串的比较

图2-1 C语言与SDS字符串的比较

C 字符串	SDS
获取字符串长度的复杂度为 $O(N)$	获取字符串长度的复杂度为 $O(1)$
API 是不安全的，可能会造成缓冲区溢出	API 是安全的，不会造成缓冲区溢出
修改字符串长度 N 次必然需要执行 N 次内存重分配	修改字符串长度 N 次最多需要执行 N 次内存重分配
只能保存文本数据	可以保存文本或者二进制数据
可以使用所有 <string.h> 库中的函数	可以使用一部分 <string.h> 库中的函数

2.3 SDS API

图 2-2 SDS 常用 API

图 2-2 SDS 常用 API

函 数	作 用	时间复杂度
sdsnew	创建一个包含给定 C 字符串的 SDS	$O(N)$, N 为给定 C 字符串的长度
sdsempty	创建一个不包含任何内容的空 SDS	$O(1)$
sdsfree	释放给定的 SDS	$O(N)$, N 为被释放 SDS 的长度
sdslen	返回 SDS 的已使用空间字节数	这个值可以通过读取 SDS 的 <code>len</code> 属性来直接获得, 复杂度为 $O(1)$
sdsavail	返回 SDS 的未使用空间字节数	这个值可以通过读取 SDS 的 <code>free</code> 属性来直接获得, 复杂度为 $O(1)$
sdsdup	创建一个给定 SDS 的副本 (copy)	$O(N)$, N 为给定 SDS 的长度
sdsclr	清空 SDS 保存的字符串内容	因为惰性空间释放策略, 复杂度为 $O(1)$
sdsconcat	将给定 C 字符串拼接到 SDS 字符串的末尾	$O(N)$, N 为被拼接 C 字符串的长度
sdsconcatSDS	将给定 SDS 字符串拼接到另一个 SDS 字符串的末尾	$O(N)$, N 为被拼接 SDS 字符串的长度

(续)

函 数	作 用	时间复杂度
sdsncpy	将给定的 C 字符串复制到 SDS 里面, 覆盖 SDS 原有的字符串	$O(N)$, N 为被复制 C 字符串的长度
sdsgrowzero	用空字符将 SDS 扩展至给定长度	$O(N)$, N 为扩展新增的字节数
sdsrange	保留 SDS 给定区间内的数据, 不在区间内的数据会被覆盖或删除	$O(N)$, N 为被保留数据的字节数
sdsstrim	接受一个 SDS 和一个 C 字符串作为参数, 从 SDS 中移除所有在 C 字符串中出现过的字符	$O(N^2)$, N 为给定 C 字符串的长度
sdsncmp	对比两个 SDS 字符串是否相同	$O(N)$, N 为两个 SDS 中较短的那个 SDS 的长度

2.4 Redis

·Redis는 C로 작성된 Redis의 SDS(Simple Dynamic String)을 사용한다.

·C의 SDS는 다음과 같다.

1. SDS는 문자열을 저장하는 구조체이다.

2. SDS는 문자열의 길이를 저장한다.

3. SDS는 문자열의 인덱스를 저장한다.

4. SDS는 문자열의 포인터를 저장한다.

5. SDS는 C의 문자열을 저장한다.

2.5 字符串

- C 字符串库函数在 15 和 16 号头文件中定义，SDS 字符串库函数在 17 号头文件中定义。

- 二进制安全 Binary Safe <http://en.wikipedia.org/wiki/Binary-safe> <http://computer.yourdictionary.com/binary-safe> 字符串库函数。

- 空字符串 Null-terminated string 字符串库函数在 18 号头文件中定义，C 字符串库函数在 19 号头文件中定义。
http://en.wikipedia.org/wiki/Null-terminated_string

- C 字符串库函数 14 号头文件中定义 API 字符串库函数 API 字符串库函数

- GNU C 字符串库函数 GNU C 字符串库函数/string 字符串库函数 API 字符串库函数 <http://www.gnu.org/software/libc>

3. 列表

Redis 列表（list）是 Redis 的四大基本数据类型之一，是一个有序的字符串集合。列表中的元素按照插入的顺序排列，你可以添加、删除列表的元素。列表是双向链表结构，每个元素都是一个字符串。

Redis 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。Redis 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。

Redis 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。Redis 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。

Redis 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。

```
redis> LLEN integers
(integer) 1024
redis> LRANGE integers 0 10
1)"1"
2)"2"
3)"3"
4)"4"
5)"5"
6)"6"
7)"7"
8)"8"
9)"9"
10)"10"
11)"11"
```

integers 列表的底层实现是 Redis 的 C 语言实现，使用双向链表结构。

Redis 的 output buffer 在 Redis 中是一个全局的缓冲区，用于存储 Redis 的响应数据。它的大小可以通过配置项 `maxmemory` 来设置。默认情况下，Redis 的 `maxmemory` 设置为 0，表示没有限制。但是，如果 Redis 的内存使用量超过了这个限制，Redis 就会开始清理数据，以防止内存溢出。

Redis 的 API 接口

Redis 的 API 接口可以分为两部分：客户端接口和服务器接口。客户端接口用于客户端与服务器之间的通信，而服务器接口用于服务器内部的通信。Redis 的客户端接口包括 `redis-cli`、`redis-py`、`redis-cpp` 等。服务器接口包括 `redis-server`、`redis-trib` 等。Redis 的 API 接口在不同的版本中可能会有所变化，因此在使用时需要参考相应的文档。

3.1 双向链表

双向链表的头文件 `adlist.h/listNode`

```
typedef struct listNode {  
    //  
    struct listNode * prev;  
    //  
    struct listNode * next;  
    //  
    void * value;  
}listNode;
```

双向链表的 `listNode` 结构体 `prev` `next` 指针指向 3-1 图

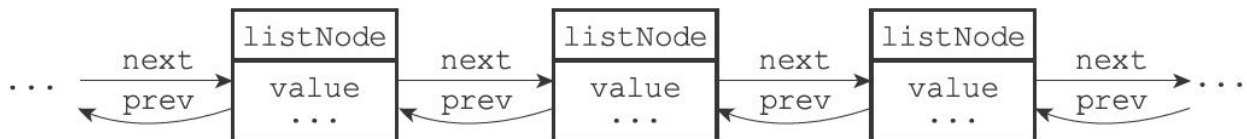


图 3-1 双向链表的 `listNode` 结构体

双向链表的头文件 `adlist.h/list`

双向链表

```
typedef struct list {  
    //  
    listNode * head;  
    //  
    listNode * tail;
```

```

//
// unsigned long len;
//
// void *(*dup)(void *ptr);
//
// void (*free)(void *ptr);
//
// int (*match)(void *ptr,void *key);
} list;

```

list 结构体成员: head, tail, len, dup, free, match

·dup 指向一个函数指针

·free 指向一个函数指针

·match 指向一个函数指针

图 3-2 list 结构体与 listNode 结构体

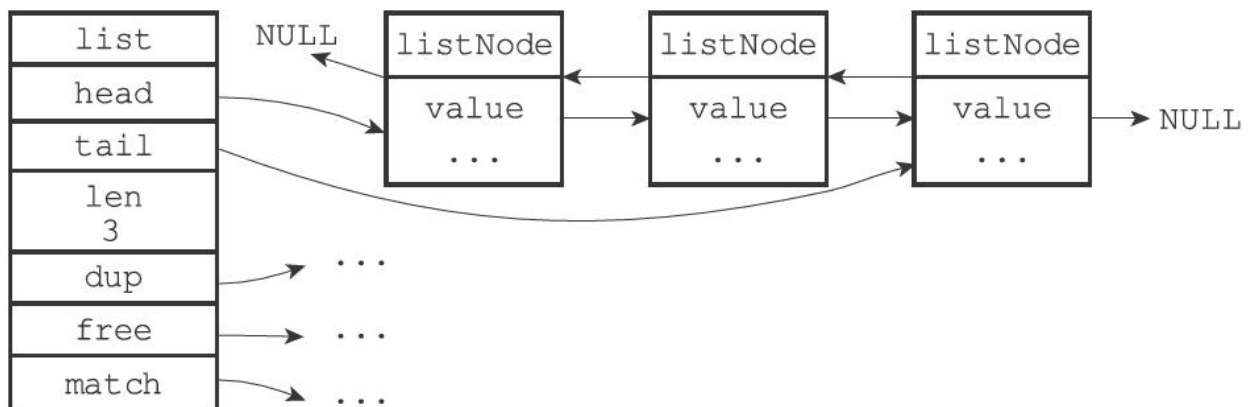


图 3-2 list 结构体与 listNode 结构体

Redis list.h

· list_node_t prev list_node_t next void

 1

· list_node_t prev list_node_t next void NULL void NULL void

 1

· list list head list tail void

 0 1

· list len list

 0 1

· void* list dup void free match

 1

3.2 链表API

图3-1 链表API

图3-1 链表API

函数	作用	时间复杂度
listSetDupMethod	将给定的函数设置为链表的节点值复制函数	复制函数可以通过链表的 dup 属性直接获得, $O(1)$
listGetDupMethod	返回链表当前正在使用的节点值复制函数	$O(1)$
listSetFreeMethod	将给定的函数设置为链表的节点值释放函数	释放函数可以通过链表的 free 属性直接获得, $O(1)$
listGetFree	返回链表当前正在使用的节点值释放函数	$O(1)$
listSetMatchMethod	将给定的函数设置为链表的节点值对比函数	对比函数可以通过链表的 match 属性直接获得, $O(1)$
listGetMatchMethod	返回链表当前正在使用的节点值对比函数	$O(1)$

(续)

函数	作用	时间复杂度
listLength	返回链表的长度 (包含了多少个节点)	链表长度可以通过链表的 len 属性直接获得, $O(1)$
listFirst	返回链表的表头节点	表头节点可以通过链表的 head 属性直接获得, $O(1)$
listLast	返回链表的表尾节点	表尾节点可以通过链表的 tail 属性直接获得, $O(1)$
listPrevNode	返回给定节点的前置节点	前置节点可以通过节点的 prev 属性直接获得, $O(1)$
listNextNode	返回给定节点的后置节点	后置节点可以通过节点的 next 属性直接获得, $O(1)$
listNodeValue	返回给定节点目前正在保存的值	节点值可以通过节点的 value 属性直接获得, $O(1)$
listCreate	创建一个不包含任何节点的新链表	$O(1)$

listAddNodeHead	将一个包含给定值的新节点添加到给定链表的表头	$O(1)$
listAddNodeTail	将一个包含给定值的新节点添加到给定链表的表尾	$O(1)$
listInsertNode	将一个包含给定值的新节点添加到给定节点的之前或者之后	$O(1)$
listSearchKey	查找并返回链表中包含给定值的节点	$O(N)$, N 为链表长度
listIndex	返回链表在给定索引上的节点	$O(N)$, N 为链表长度
listDelNode	从链表中删除给定节点	$O(N)$, N 为链表长度
listRotate	将链表的表尾节点弹出, 然后将被弹出的节点插入到链表的表头, 成为新的表头节点	$O(1)$
listDup	复制一个给定链表的副本	$O(N)$, N 为链表长度
listRelease	释放给定链表, 以及链表中的所有节点	$O(N)$, N 为链表长度

3.3 实现

- 实现 Redis 的 `list` 命令

- 实现 `listNode` 结构体

Redis 的 `list` 命令

- 实现 `list` 命令

- 实现 `listNode` 结构体

实现

- 实现 Redis 的 `list` 命令

4 章

symbol table associative array
map key-value pair

key value
map

map
map

Redis C
Redis

Redis Redis
Redis

Redis

```
redis> SET msg "hello world"
OK
```

"msg" "hello world"
Redis

redis> HSETALL website 10086 "Redis" "Redis.io" "MariaDB" "MariaDB.org" "MongoDB" "MongoDB.org"

redis> HLEN website
(integer) 10086
redis> HGETALL website
1)"Redis"
2)"Redis.io"
3)"MariaDB"
4)"MariaDB.org"
5)"MongoDB"
6)"MongoDB.org"
...

redis> HSETALL website 10086 "Redis" "Redis.io" "MariaDB" "MariaDB.org" "MongoDB" "MongoDB.org"

redis> HSETALL website 10086 "Redis" "Redis.io" "MariaDB" "MariaDB.org" "MongoDB" "MongoDB.org"

redis> HSETALL website 10086 "Redis" "Redis.io" "MariaDB" "MariaDB.org" "MongoDB" "MongoDB.org"

redis> HSETALL website 10086 "Redis" "Redis.io" "MariaDB" "MariaDB.org" "MongoDB" "MongoDB.org"

- `Associative Array`

`http://en.wikipedia.org/wiki/Associative_array` `Hash Table`

`http://en.wikipedia.org/wiki/Hash_table`

- `C` `14`

- `11`

4.1 字典

Redis 字典是 Redis 中存储键值对数据的数据结构。字典是 Redis 中最重要的数据结构之一，它用于存储键值对数据。字典的实现基于哈希表，它允许在 O(1) 的时间复杂度内查找、插入和删除元素。

字典的实现依赖于 Redis 的字典库，该库位于 `dict.h` 文件中。

4.1.1 字典结构

Redis 字典结构定义在 `dict.h` 文件中，如下所示：

```
typedef struct dictht {  
    //  
    dictEntry **table;  
    //  
    unsigned long size;  
    //  
    unsigned long sizemask;  
    //  
    unsigned long used;  
} dictht;
```

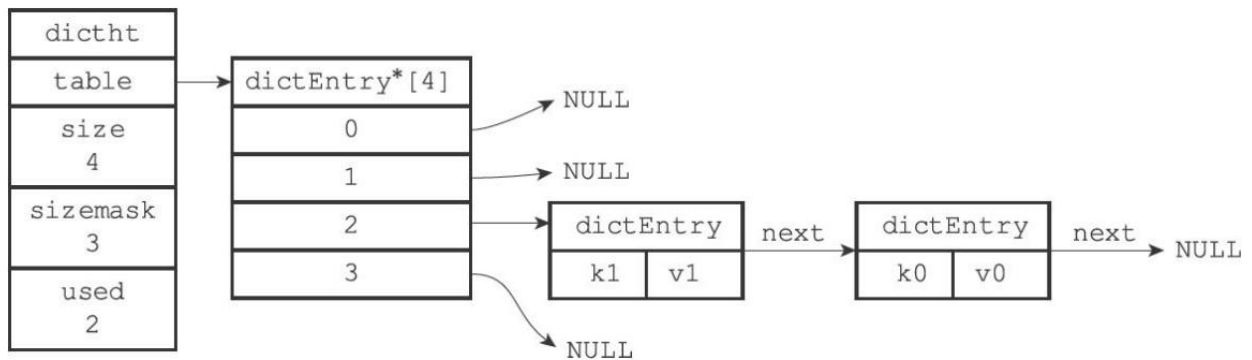
字典的 `table` 成员是一个指向 `dictEntry` 结构的指针数组，该数组的大小为 `size`。字典的 `size` 成员表示字典中当前存储的元素数量，而 `sizemask` 成员则表示字典的大小掩码。字典的 `used` 成员表示字典中当前使用的槽位数量。


```
//
// 字典链表的字典项
struct dictEntry *next;
} dictEntry;
```

key 字典项的键 v 字典项的值
 uint64_t 字典项的键 int64_t 字典项的值

next 字典项的下一个字典项
 collision 字典项的碰撞

字典项 4-2 字典项的下一个字典项 k1 k0



4-2 字典项的下一个字典项 k1 k0

4.1.3 字典

Redis 字典 dict.h/dict

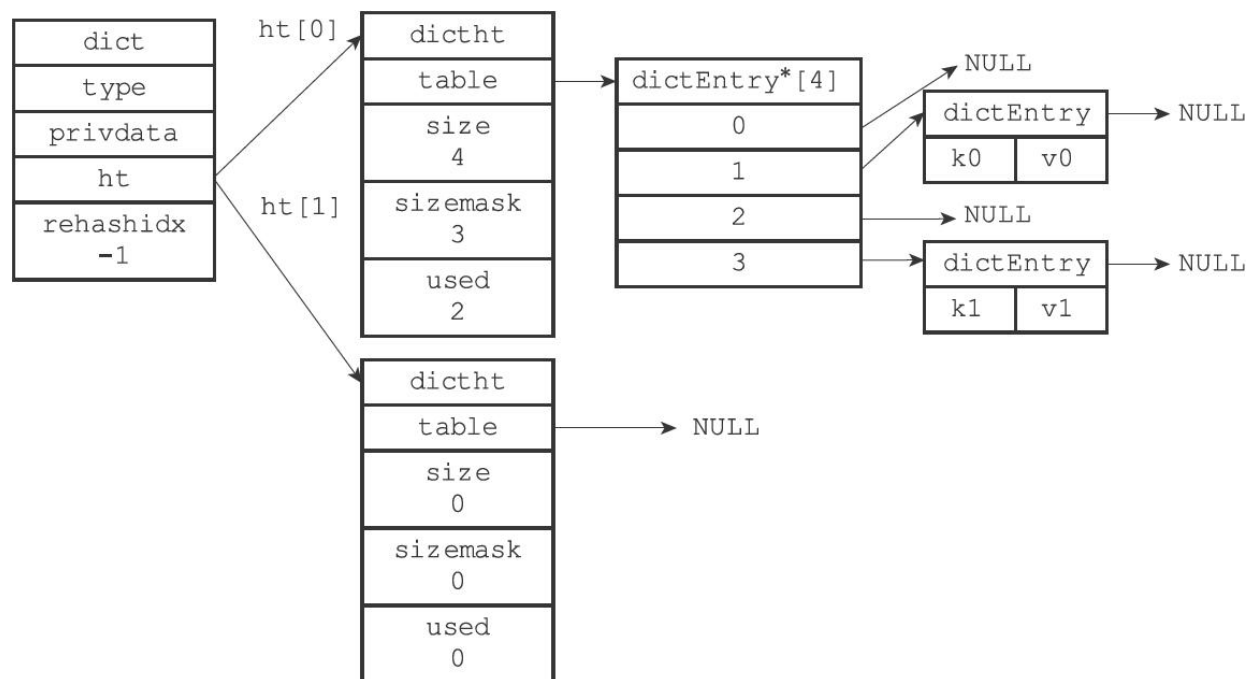
```
typedef struct dict {
    //
    dictType *type;
    //
```


ht[] dict ht[]

ht[0] ht[1] ht[0] rehash

ht[1] rehash rehashidx rehash
rehash-1

4-3 rehash

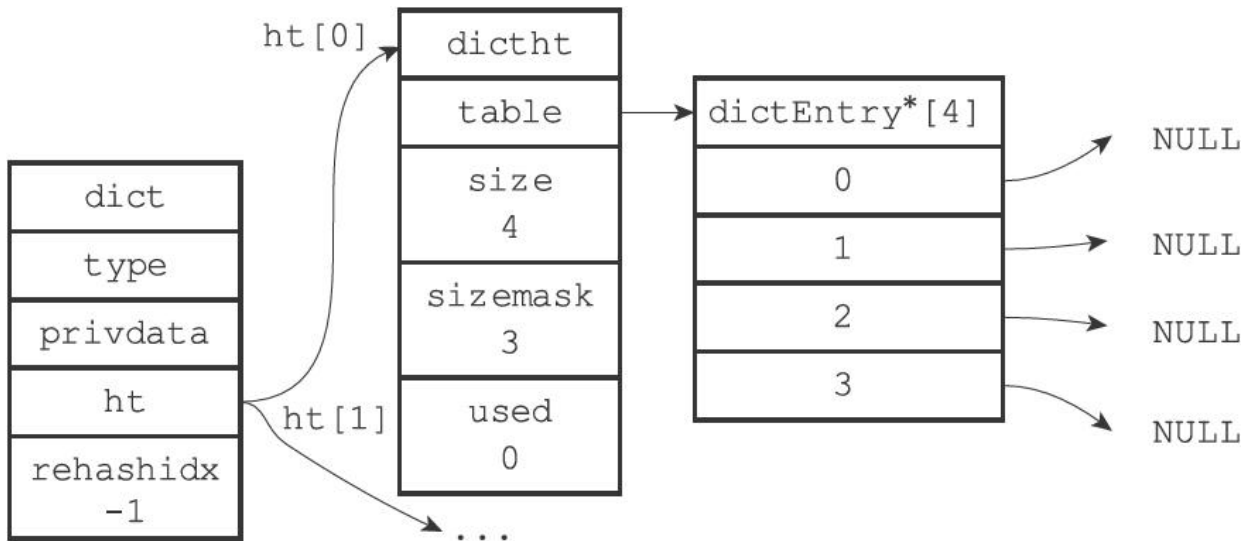


4-3

4.2 □□□□

[illegible]Redis

```
#
key
hash = dict->type->hashFunction(key);
#
sizemask
#
ht[x]
ht[0]
ht[1]
index = hash & dict->ht[x].sizemask;
```



□4-4 □□□

4-4 字典的哈希函数 k0 v0

```
hash = dict->type->hashFunction(k0);
```

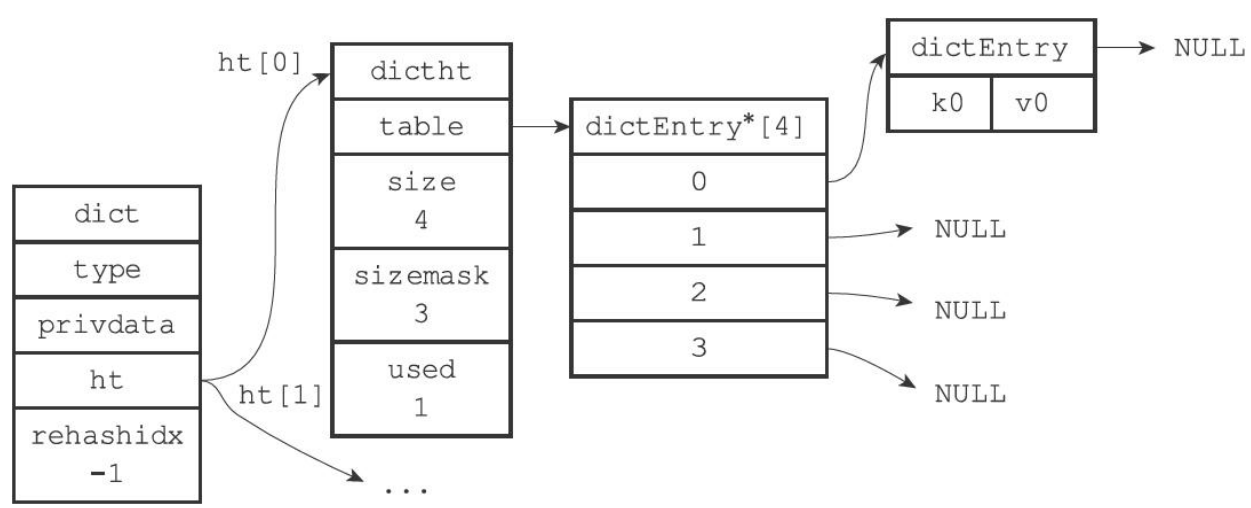
k0

8

```
index = hash&dict->ht[0].sizemask = 8 & 3 = 0;
```

k0 0 k0 v0 0

4-5



4-5 字典的哈希函数 k0 v0

Redis MurmurHash2

[illegible]

MurmurHash2 MurmurHash

<http://code.google.com/p/smhasher/>

4.3 字典

字典是Redis中最重要的数据结构之一，它用于存储键值对。

字典的碰撞（collision）

Redis字典使用separate chaining（分离链）来处理碰撞。当多个键映射到同一个桶（bucket）时，这些键值对会被存储在桶的链表中。每个字典项（dictEntry）都有一个next指针，指向链中的下一个字典项。如果next指针为NULL，表示该桶中的链已经结束。

字典的扩容（resize）操作。当字典中的键值对数量增加，导致碰撞率（collision rate）超过一定阈值（通常是4-6）时，字典需要进行扩容。扩容过程中，字典会创建一个新的更大的桶数组，并将旧字典中的键值对重新哈希并插入到新桶数组中。这个过程是渐进式的，不会阻塞其他操作。

字典的字典项（dictEntry）结构。每个字典项包含键（key）和值（value）。字典项的键和值可以是任意类型的Redis对象。字典项的next指针指向链中的下一个字典项。字典项的哈希值（hash）存储在字典项的k0或k1字段中，具体取决于字典项是否被哈希过。

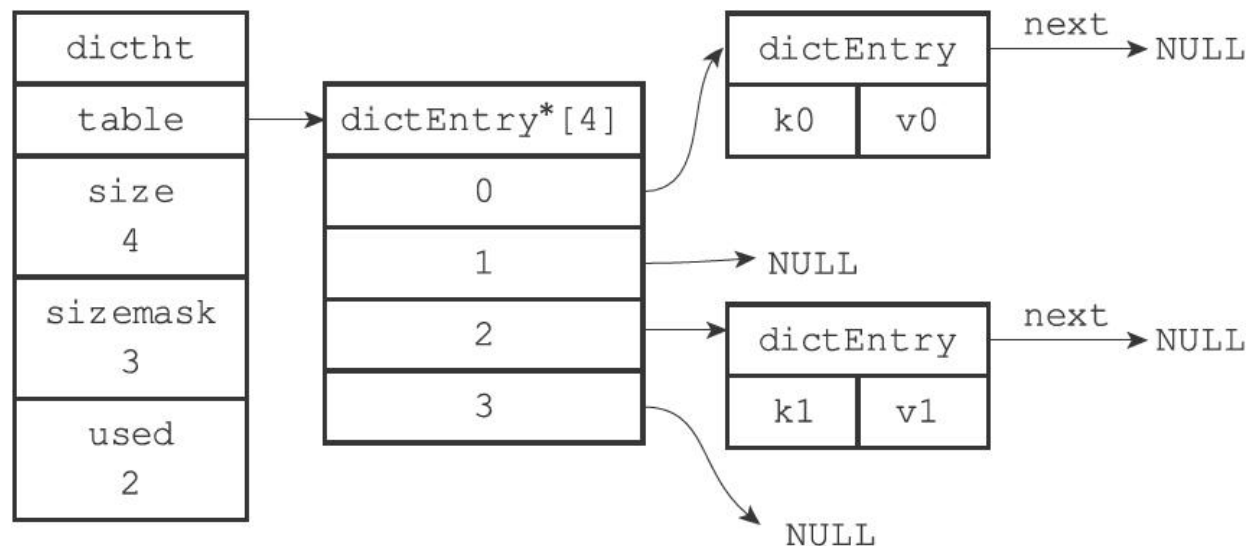


图4-6 字典数据结构

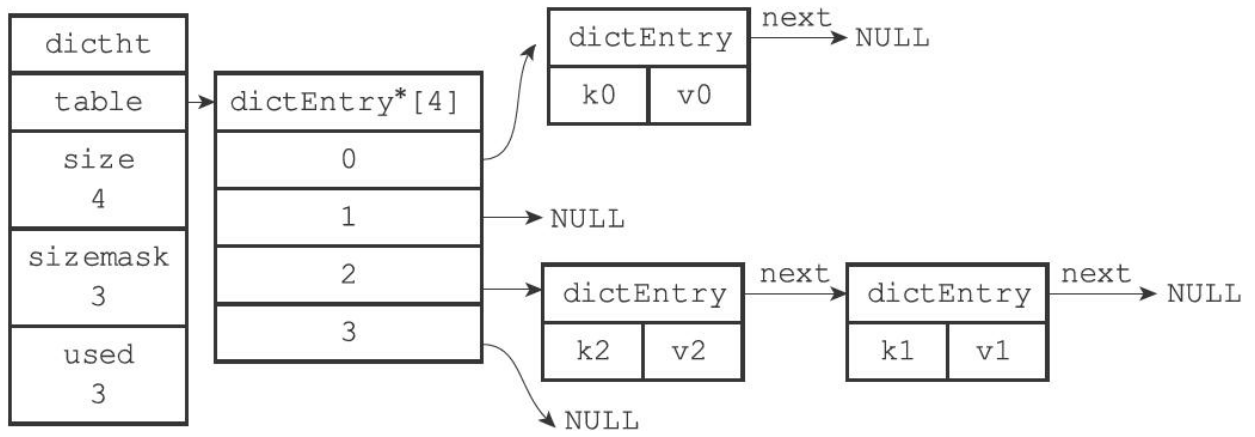


图4-7 字典数据结构

4.4 rehash

Redis 的字典使用两个哈希表实现。当字典的负载因子（load factor）达到一定阈值时，就需要进行 rehash 操作，将数据从一个哈希表迁移到另一个哈希表。

Redis 的 rehash 操作是在后台线程中进行的，不会阻塞主线程。rehash 操作的具体步骤如下：

1. 创建一个新的哈希表（ht[1]），其容量是旧哈希表（ht[0]）容量的两倍。新哈希表的 used 属性初始化为 0。

2. 遍历旧哈希表（ht[0]）中的所有元素，将每个元素迁移到新哈希表（ht[1]）。迁移过程中，旧哈希表的 used 属性会增加。

3. 当旧哈希表（ht[0]）中的所有元素都迁移到新哈希表（ht[1]）后，旧哈希表的 used 属性会重置为 0。

4. 重复上述步骤，直到旧哈希表（ht[0]）中的所有元素都迁移到新哈希表（ht[1]）后，旧哈希表的 used 属性会重置为 0。

5. 当旧哈希表（ht[0]）中的所有元素都迁移到新哈希表（ht[1]）后，旧哈希表的 used 属性会重置为 0。

Redis 的 rehash 操作会在后台线程中定期进行，通常每 4-8 秒进行一次。

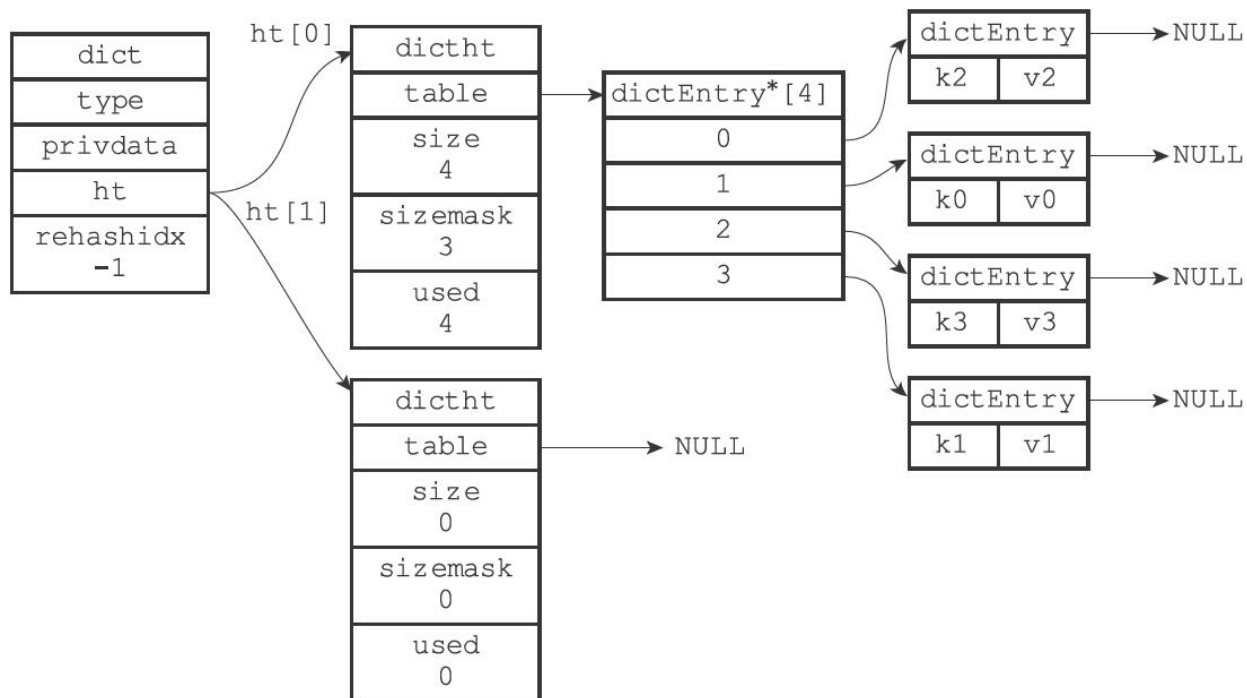


图4-8 字典rehash过程

1. `ht[0].used` 为 4， $4 \times 2 = 8$ ， 8×2^3 为 64，即新表大小为 64。n 为 64。

2. `ht[1]` 为 8，4-9 为 `ht[1]` 的索引范围。

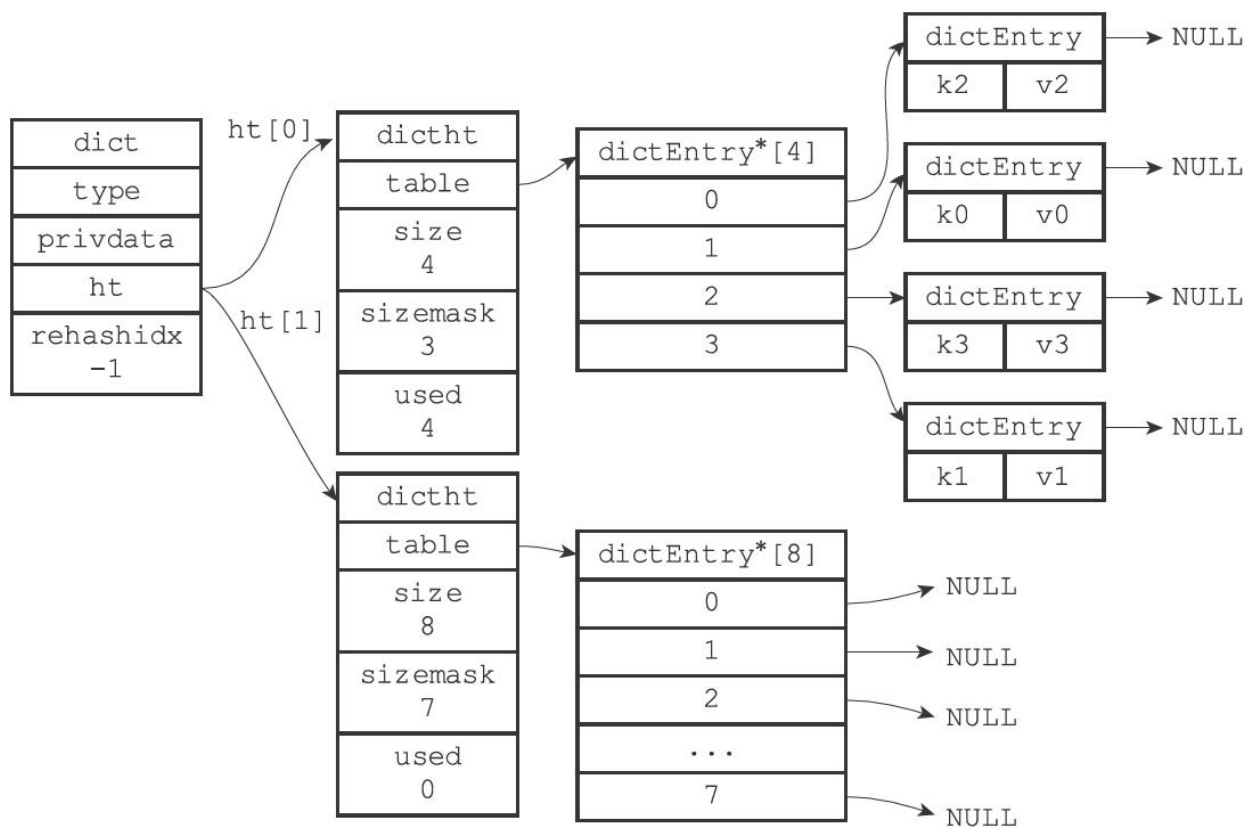


图4-9 字典结构图

字典结构图

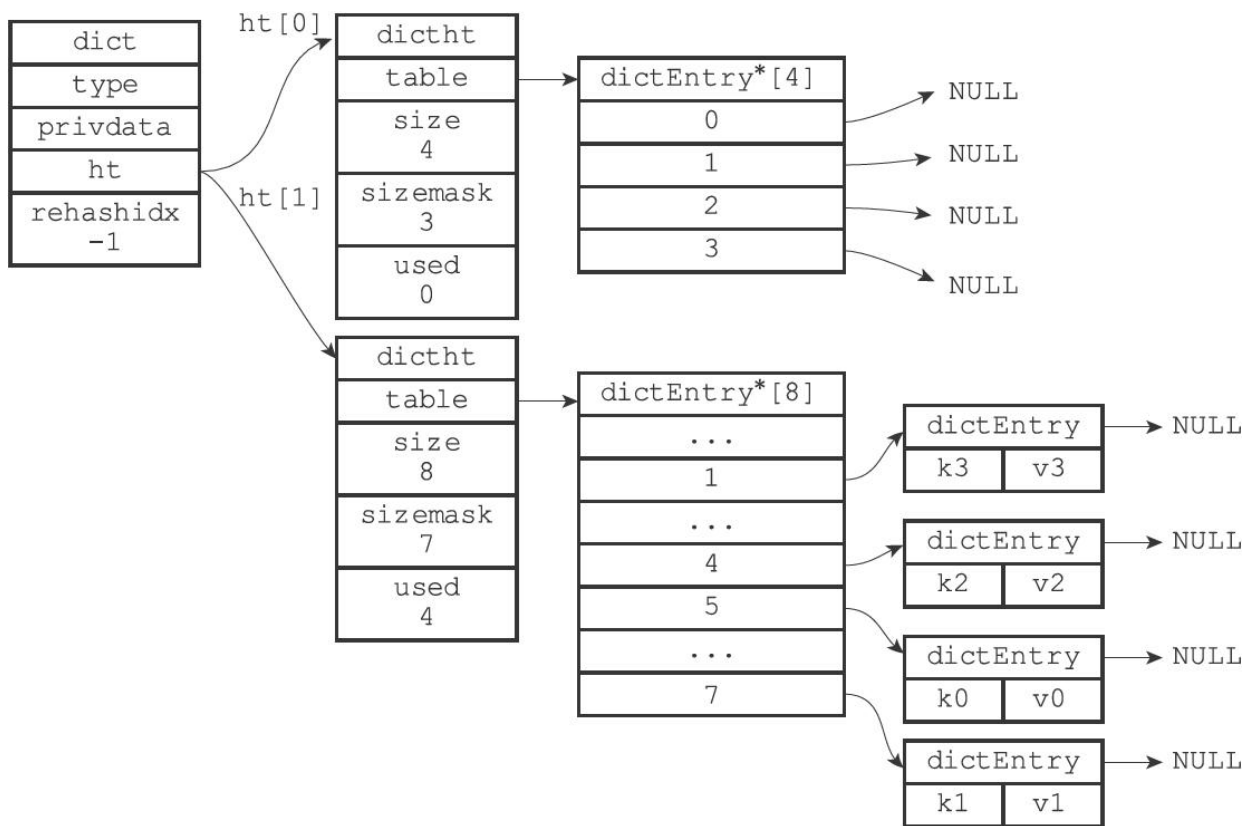


图4-10 ht[0]和ht[1]的指向

3. 字典的哈希表（ht）的初始化（ht[0]和ht[1]）

字典的哈希表（ht）的初始化（ht[0]和ht[1]）的初始化过程如图4-11所示。

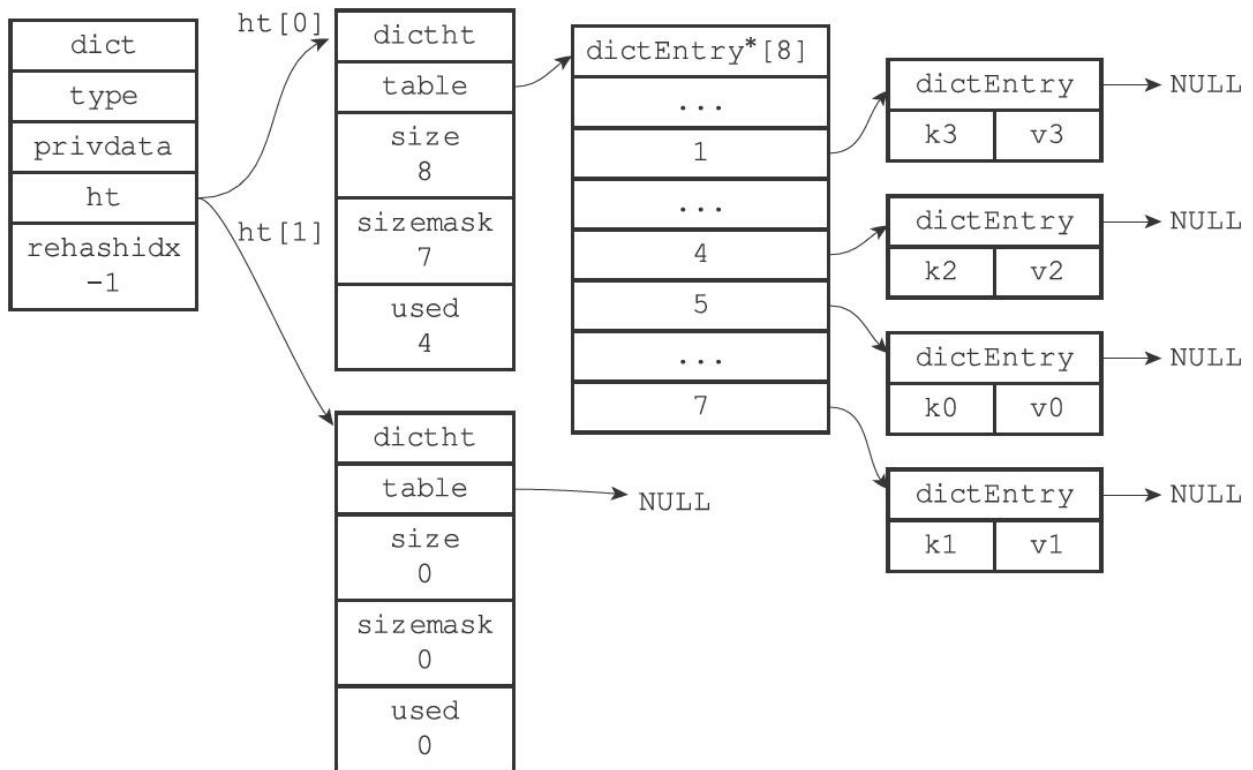


图4-11 rehash过程

操作步骤

1. 初始化 rehash 过程

1. 调用 `BGSAVE` 或 `BGREWRITEAOF` 进行持久化
 调用 `1`

2. 调用 `BGSAVE` 或 `BGREWRITEAOF` 进行持久化
 调用 `5`

初始化 rehash 过程

```
#
load_factor =
load_factor /
load_factor
load_factor = ht[0].used / ht[0].size
```

load_factor

load_factor = 4 / 4 = 1

load_factor = 4 / 4 = 1

load_factor = 512 / 256 = 2

load_factor = 256 / 512 = 0.5

BGSAVE BGREWRITEAOF Redis copy-on-write

load_factor = 0.1

4.5 重新hash

如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。

如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。如果桶 `ht[1]` 中的元素太多，那么可以重新hash桶 `ht[0]` 中的元素到桶 `ht[1]` 中。

如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。如果桶 `ht[1]` 中的元素太多，那么可以重新hash桶 `ht[0]` 中的元素到桶 `ht[1]` 中。

如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。

1. 如果桶 `ht[1]` 中的元素太多，那么可以重新hash桶 `ht[0]` 中的元素到桶 `ht[1]` 中。

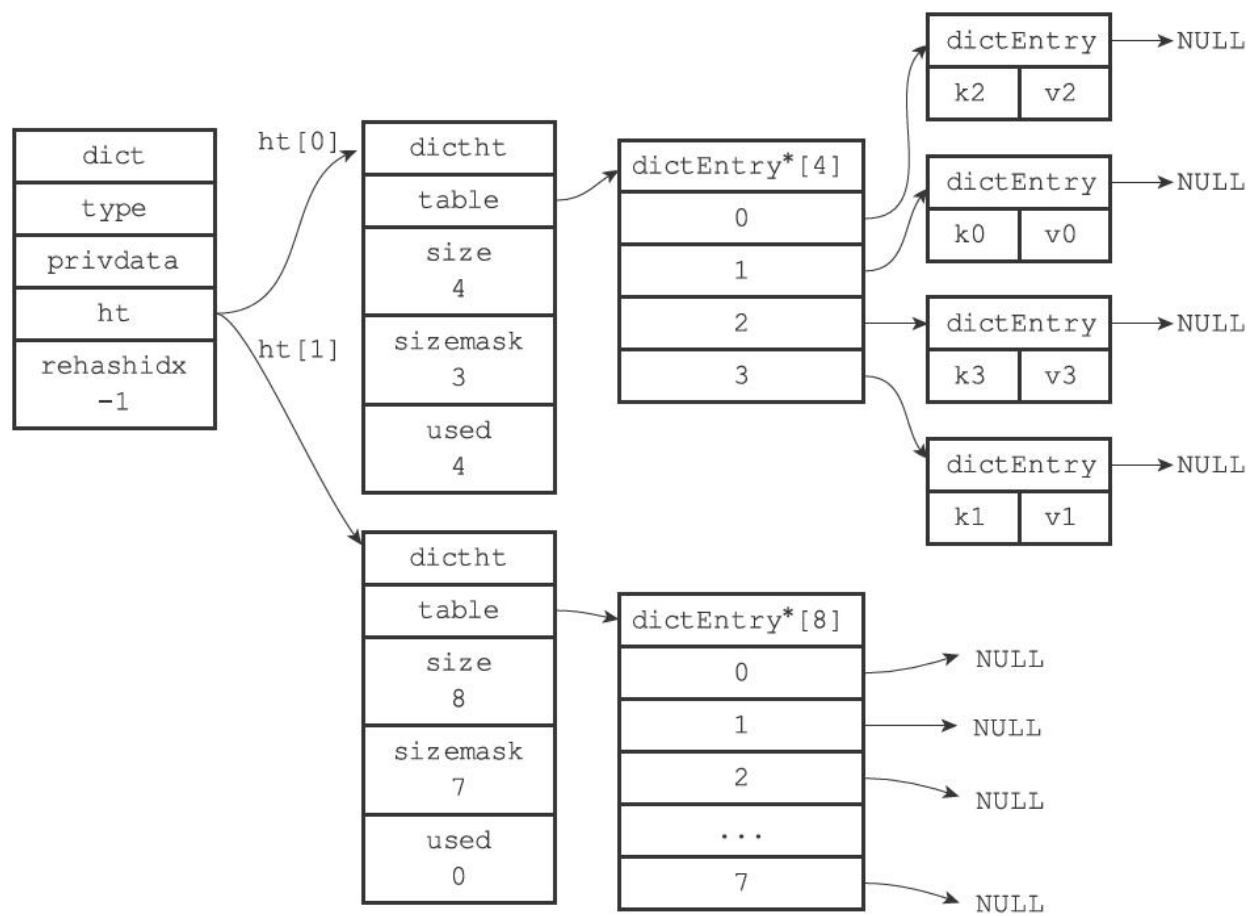
2. 如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。

3. 如果桶 `ht[0]` 中的元素太多，那么可以重新hash桶 `ht[1]` 中的元素到桶 `ht[0]` 中。如果桶 `ht[1]` 中的元素太多，那么可以重新hash桶 `ht[0]` 中的元素到桶 `ht[1]` 中。

4个字典头指针ht[0]指向的字典头
 ht[1]指向的字典头rehashidx-1指向的字典头

字典头rehash指向的字典头rehash指向的字典头
 rehash指向的字典头rehash指向的字典头

4-12 4-17 字典头rehash指向的字典头rehash指向的字典头
 rehashidx指向的字典头



4-12 字典头rehash

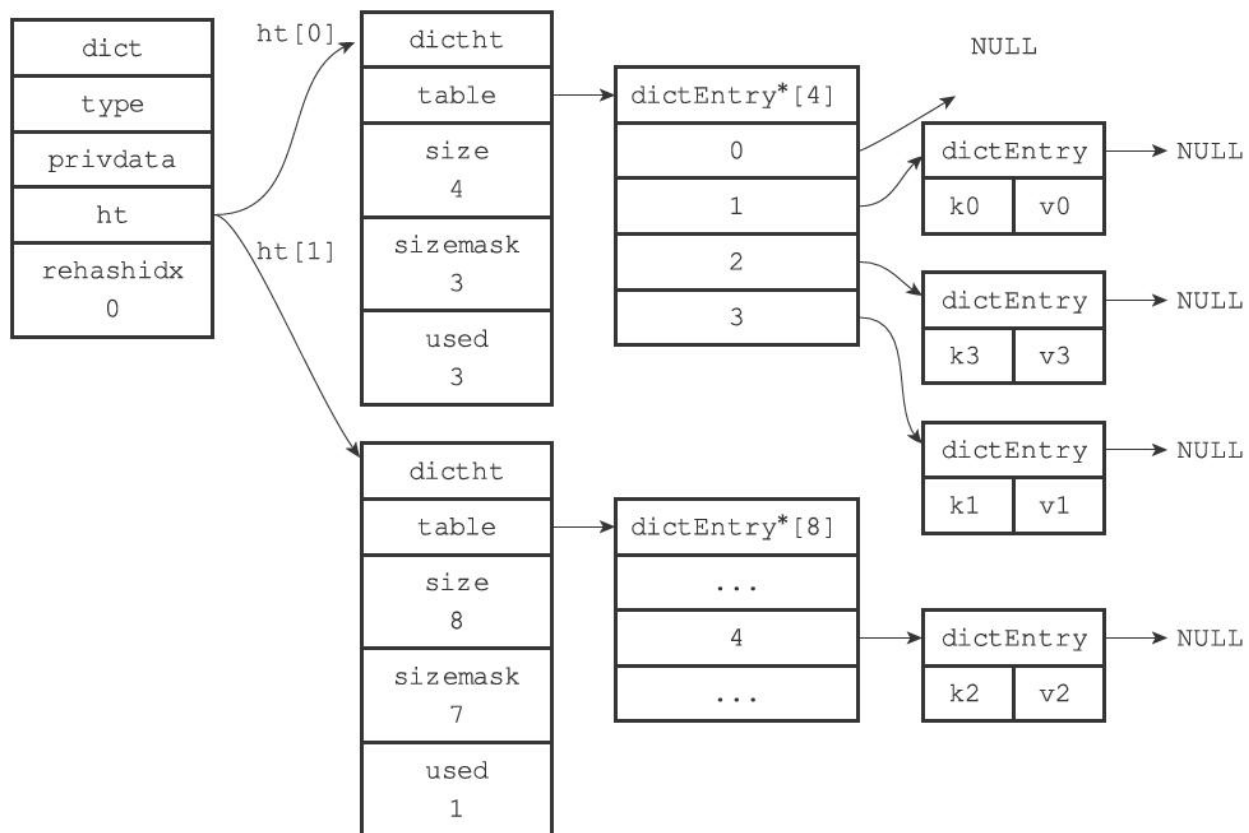


图4-13 rehash过程示意图

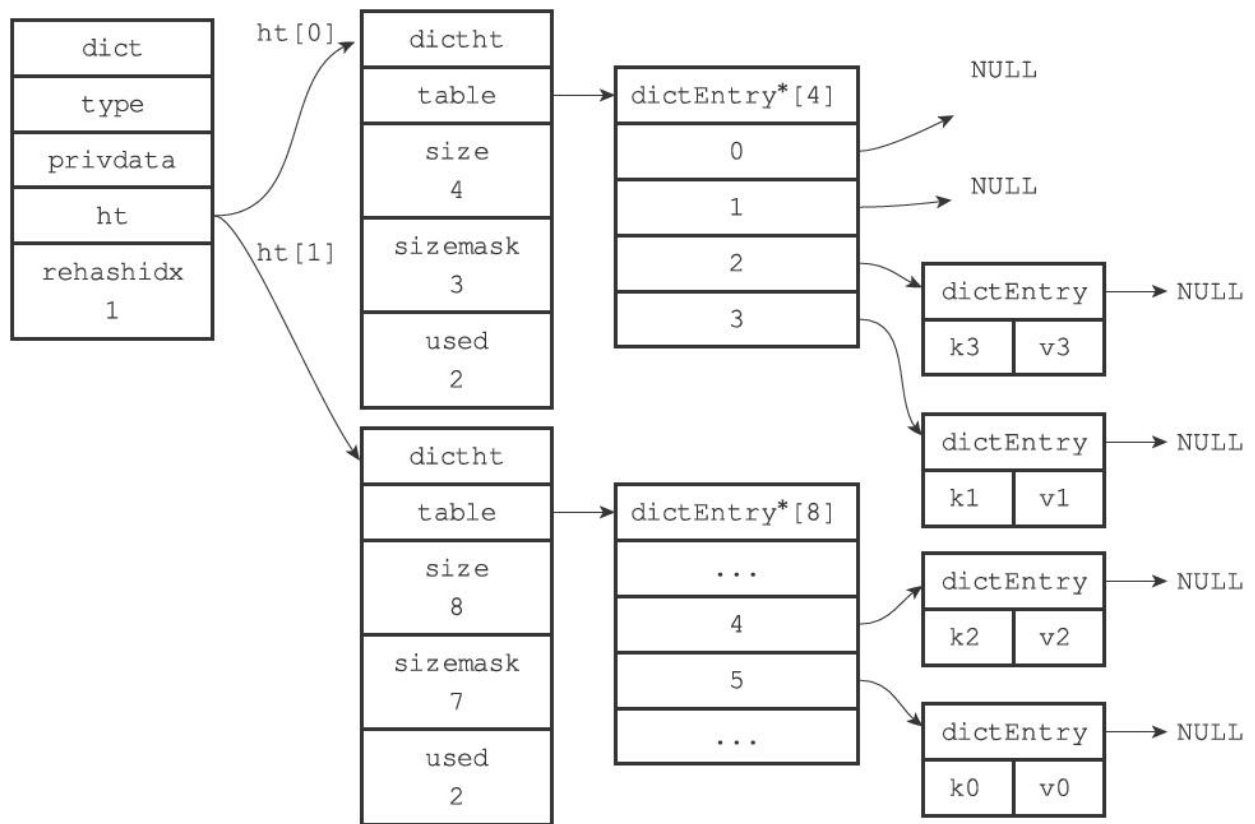
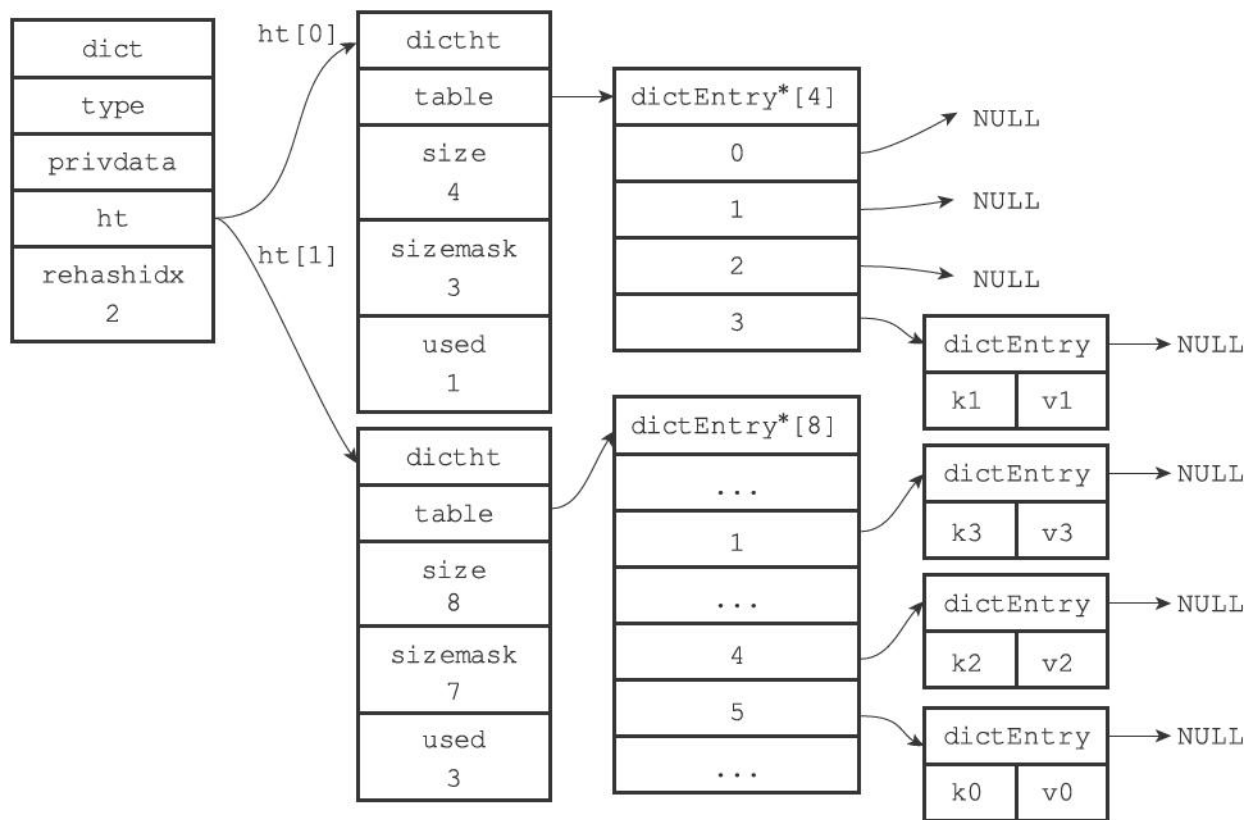


图4-14 rehash过程示意图



4-15 rehash 2

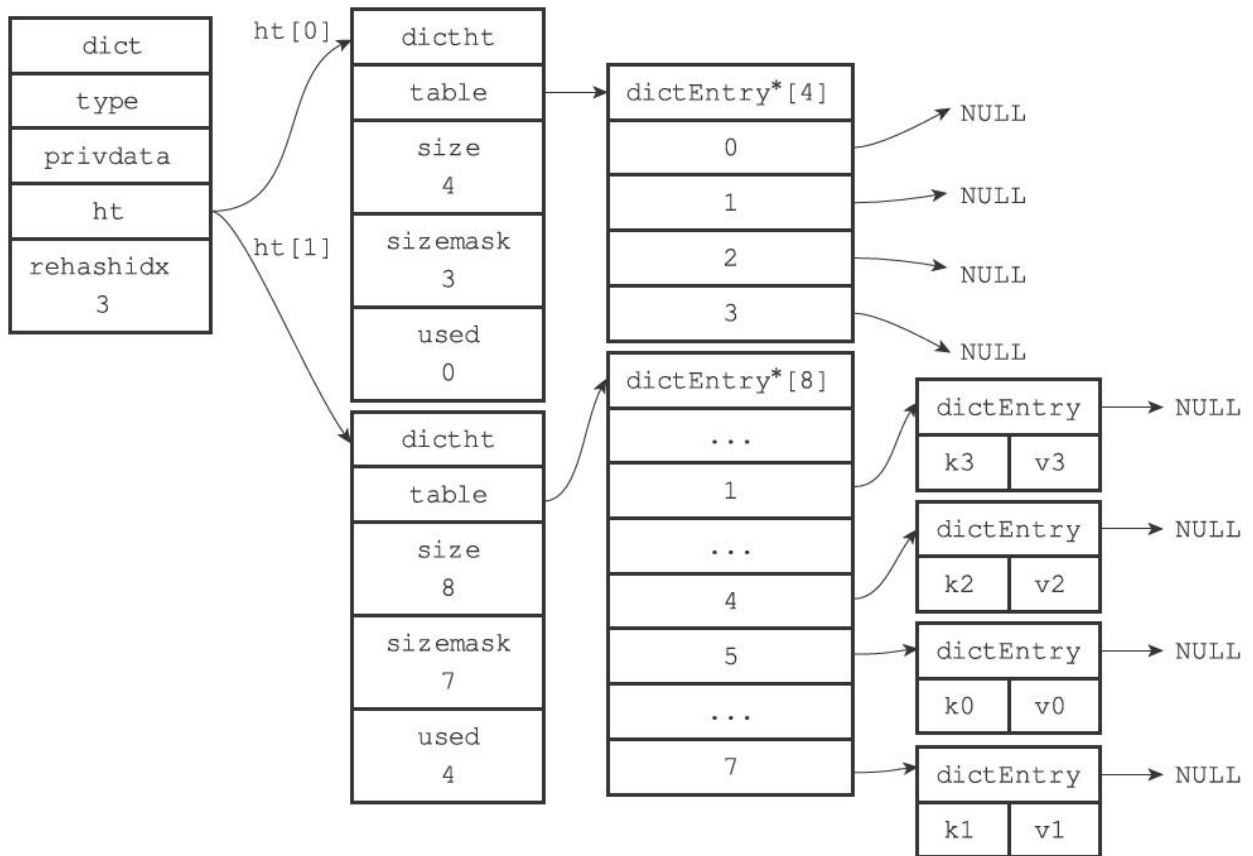


图4-16 rehash前

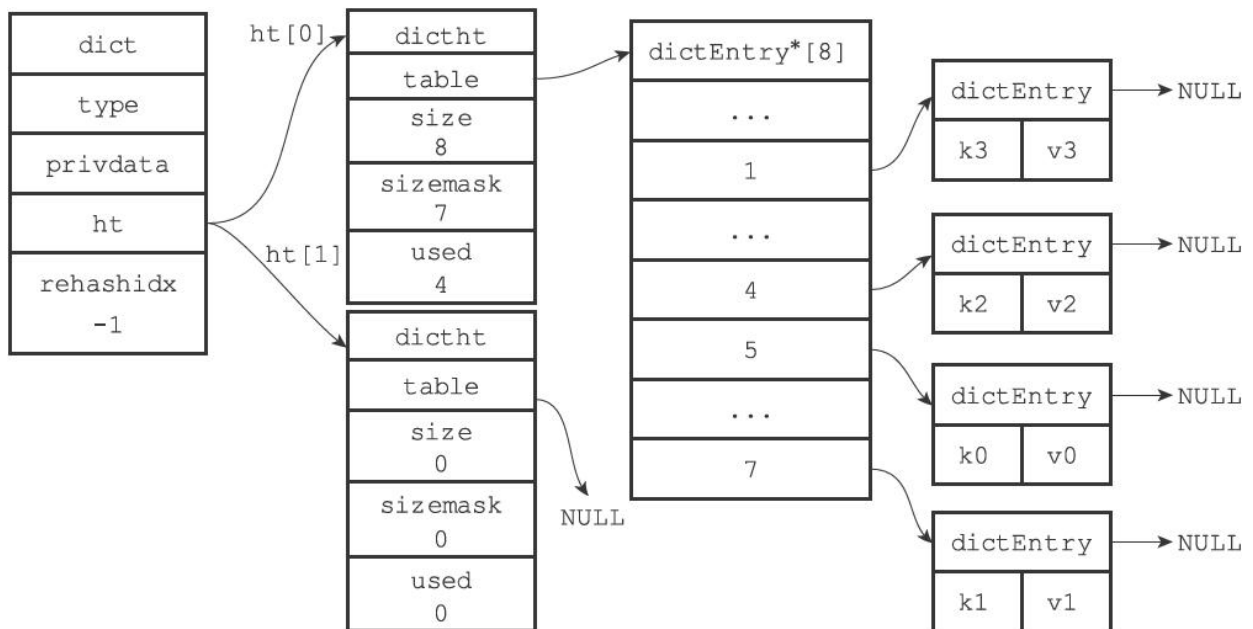


图4-17 rehash中

rehash

rehash
delete
find
update
ht[0]
ht[1]

rehash
ht[1]
ht[0]
rehash

4.6 字典API

表4-1 字典API

表4-1 字典API

函 数	作 用	时间复杂度
dictCreate	创建一个新的字典	$O(1)$
dictAdd	将给定的键值对添加到字典里面	$O(1)$
dictReplace	将给定的键值对添加到字典里面，如果键已经存在于字典，那么用新值取代原有的值	$O(1)$
dictFetchValue	返回给定键的值	$O(1)$
dictGetRandomKey	从字典中随机返回一个键值对	$O(1)$

(续)

函 数	作 用	时间复杂度
dictDelete	从字典中删除给定键所对应的键值对	$O(1)$
dictRelease	释放给定字典，以及字典中包含的所有键值对	$O(N)$ ， N 为字典包含的键值对数量

4.7 字典

- 字典是Redis中最基本的数据类型

- Redis字典使用哈希表实现，支持多种哈希算法，如：
rehash

- Redis字典使用MurmurHash2算法进行哈希计算，
字典的哈希值

- 字典的哈希值用于快速查找字典中的元素

- 字典的哈希值用于快速查找字典中的元素，
字典的哈希值用于快速查找字典中的元素

5 题

skiplist 数据结构
时间复杂度

$O(\log N)$ 时间复杂度
空间复杂度

Redis 数据结构
时间复杂度

Redis member 数据结构
时间复杂度

fruit-price 数据结构
130 题

```
redis> ZRANGE fruit-price 0 2 WITHSCORES
1)"banana"
2)"5"
3)"cherry"
4)"6.5"
5)"apple"
6)"8"
redis> ZCARD fruit-price
(integer)130
```

```
fruit-price[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] node[] [] []  
[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []
```

```
·"banana"5
```

```
·"cherry"6.5
```

·"apple"8

Redis Redis
Redis
Redis API
WilliamPugh Skip Lists:A Probabilistic Alternative to Balanced Trees C
14 13.5

5.1 跳跃表

Redis的跳跃表在redis.h/zskiplistNode和redis.h/zskiplist中定义。跳跃表zskiplistNode和zskiplist的定义如下。

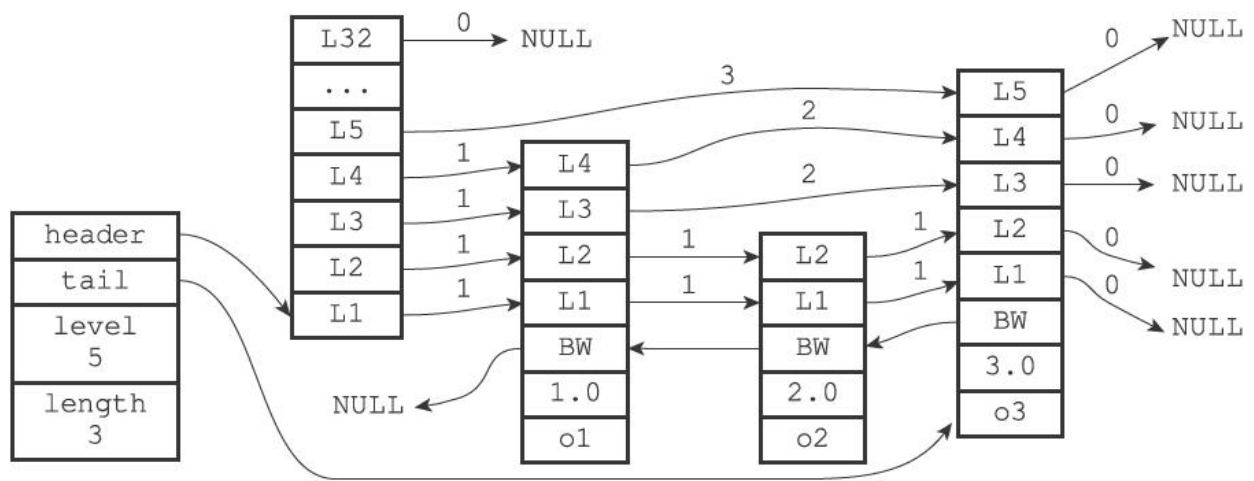


图5-1 跳跃表

图5-1展示了跳跃表zskiplist的定义。

·header指向第一个节点

·tail指向最后一个节点

·level指向当前节点所在的层数

·length指向当前节点所在层的节点个数

[illegible]

```
double score;
int
```

objSDS

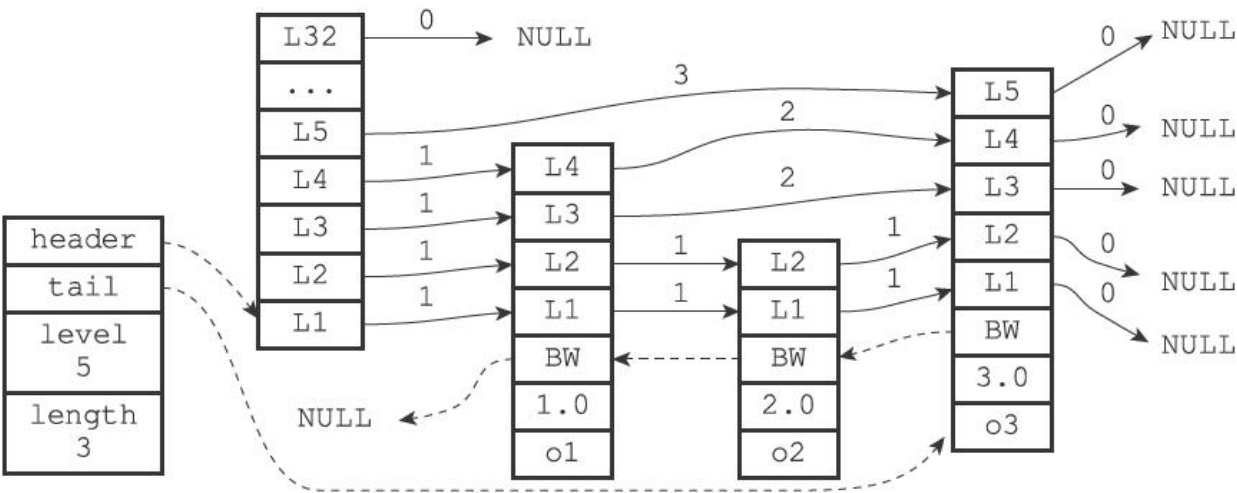


图5-6 多级链表结构

图5-7 链表节点结构

o1o2o3o2o3
o1o2o3o1<=o2<=o3

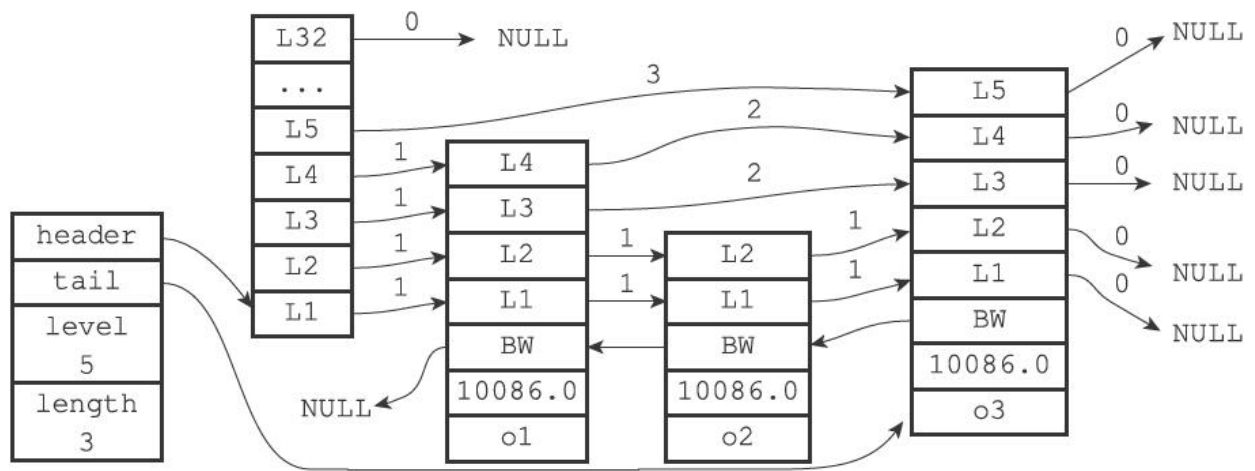


图5-7 跳表结构示意图

5.1.2 跳表

跳表是一种数据结构，其结构如图5-8所示。

跳表的头指针指向第一个节点，每个节点包含一个指向下一个节点的指针。

跳表的每个节点包含一个指向下一个节点的指针，其结构如图5-9所示。

图

跳表的头指针指向第一个节点。

```
typedef struct zskiplist {
    //
    struct zskiplistNode *header, *tail;
    //
    unsigned long length;
    //
    int level;
} zskiplist;
```


level01

5.2 跳跃表API

5-1 跳跃表API

5-1 跳跃表API

函 数	作 用	时间复杂度
<code>zslCreate</code>	创建一个新的跳跃表	$O(1)$
<code>zslFree</code>	释放给定跳跃表，以及表中包含的所有节点	$O(N)$ ， N 为跳跃表的长度
<code>zslInsert</code>	将包含给定成员和分值的新节点添加到跳跃表中	平均 $O(\log N)$ ，最坏 $O(N)$ ， N 为跳跃表长度
<code>zslDelete</code>	删除跳跃表中包含给定成员和分值的节点	平均 $O(\log N)$ ，最坏 $O(N)$ ， N 为跳跃表长度
<code>zslGetRank</code>	返回包含给定成员和分值的节点在跳跃表中的排位	平均 $O(\log N)$ ，最坏 $O(N)$ ， N 为跳跃表长度
<code>zslGetElementByRank</code>	返回跳跃表在给定排位上的节点	平均 $O(\log N)$ ，最坏 $O(N)$ ， N 为跳跃表长度
<code>zslIsInRange</code>	给定一个分值范围（ <code>range</code> ），比如 0 到 15，20 到 28，诸如此类，如果跳跃表中有至少一个节点的分值在这个范围之内，那么返回 1，否则返回 0	通过跳跃表的表头节点和表尾节点，这个检测可以用 $O(1)$ 复杂度完成
<code>zslFirstInRange</code>	给定一个分值范围，返回跳跃表中第一个符合这个范围的节点	平均 $O(\log N)$ ，最坏 $O(N)$ 。 N 为跳跃表长度
<code>zslLastInRange</code>	给定一个分值范围，返回跳跃表中最后一个符合这个范围的节点	平均 $O(\log N)$ ，最坏 $O(N)$ 。 N 为跳跃表长度
<code>zslDeleteRangeByScore</code>	给定一个分值范围，删除跳跃表中所有在这个范围之内节点	$O(N)$ ， N 为被删除节点数量
<code>zslDeleteRangeByRank</code>	给定一个排位范围，删除跳跃表中所有在这个范围之内节点	$O(N)$ ， N 为被删除节点数量

5.3 zskiplist

- zskiplistNode

- Redis 内部使用 zskiplist 结构来存储有序集合。zskiplistNode 是 zskiplist 的节点，它包含了指向下一个节点的指针，以及一个指向 zskiplist 的指针。

- zskiplistNode 的 height 属性表示该节点在 zskiplist 中的高度，高度为 1 表示该节点是 zskiplist 的叶子节点，高度为 32 表示该节点是 zskiplist 的根节点。

- zskiplistNode 的 next 属性表示指向下一个节点的指针，如果 next 为 NULL，则表示该节点是 zskiplist 的最后一个节点。

- zskiplistNode 的 value 属性表示该节点存储的值，该值是一个字节数组。

6 6.1

intset

Redis

```
redis> SADD numbers 1 3 5 7 9
(integer) 5
redis> OBJECT ENCODING numbers
"intset"
```

6.1 数据类型

在Redis中，我们使用以下数据类型：
intset、int16_t、int32_t、int64_t

在intset.h/intset.h中定义：

```
typedef struct intset {  
    //  
    uint32_t encoding;  
    //  
    uint32_t length;  
    //  
    int8_t contents[];  
} intset;
```

contents是包含所有元素的数组，其长度由length指定。

length是元素的个数，其值必须在1到255之间。

在intset.h中，我们使用以下宏来定义intset：

```
intset_t intset_new(int8_t encoding, int32_t length)
```

· encoding是元素的类型，可以是INTSET_ENC_INT16、INTSET_ENC_INT32、INTSET_ENC_INT64。

int16_t是16位的有符号整数，范围是-32768到32767。

32767

·encodingINTSET_ENC_INT32contents
int32_tint32_t
-21474836482147483647

·encodingINTSET_ENC_INT64contents
int64_tint64_t
-92233720368547758089223372036854775807

6-1



6-1 int16_t

·encodingINTSET_ENC_INT16int16_t
int16_t
·length5

·contents[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

·[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int16_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]contents[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]sizeof
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int16_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]*5=16*5=80[0][0]

图6-2 [0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

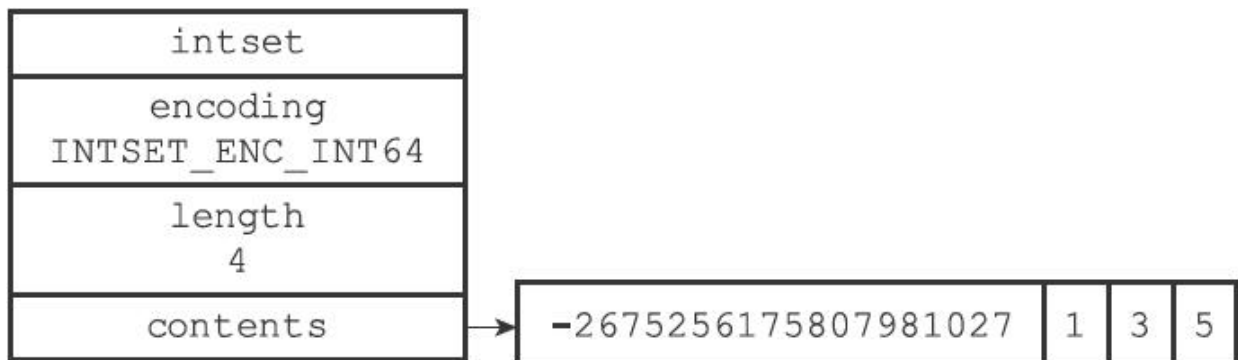


图6-2 [0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int16_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

·encoding[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]INTSET_ENC_INT64[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int64_t
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int64_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

·length[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]4[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

·contents[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

·[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int64_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]contents[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]sizeof
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int64_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]*4=64*4=256[0][0]

[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]contents[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]-2675256175807981027[0][0][0][0]
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int64_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]1[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]int16_t[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]

int16_tint64_t
int64_tcontentsint64_t
-2675256175807981027

6.2 结构

结构体定义如下，其中 `upgrade` 成员用于升级结构体。

结构体定义如下：

1. 结构体成员 `intset` 用于存储整数集合。

2. 结构体成员 `encoding` 用于存储编码，默认为 `INTSET_ENC_INT16`。

3. 结构体成员 `length` 用于存储集合中的元素个数。

结构体成员 `contents` 用于存储集合中的元素，其类型为 `int16_t`。

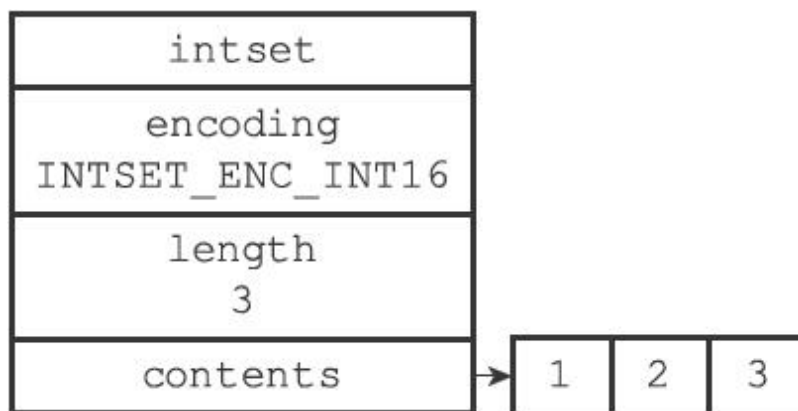


图6-3 结构体 `int16_t` 的定义

每个元素16位，3个元素共48位，即6-4所示。
每个元素48位。

位	0至15位	16至31位	32至47位
元素	1	2	3

图6-4 contents的位分配

每个元素32_t，65535，65535，
int32_t，65535，
。

每个元素，
。

每个元素65535，
int32_t，32，32*4=128，
图6-5，1，2，3，int16_t，
48，int32_t，
。

位	0至15位	16至31位	32至47位	48至127位
元素	1	2	3	(新分配空间)

图6-5 的位分配

65535 1 2 3 65535
 contents 3 96 127 6-9

位	0至31位	32至63位	64位至95位	96位至127位
元素	1	2	3	65535

↑
 添加新元素

图6-9 65535

encoding INTSET_ENC_INT16
 INTSET_ENC_INT32 length 3 4 6-10



图6-10

O(N)

INTSET_ENC_INT16
 INTSET_ENC_INT64 INTSET_ENC_INT32

INTSET_ENC_INT64

INTSET_ENC_INT64

INTSET_ENC_INT64

·INTSET_ENC_INT64

·INTSET_ENC_INT64length-1

6.3 数据类型

数据类型是C语言中非常重要的概念，它决定了变量在内存中如何存储、如何操作以及所能取值的范围。

6.3.1 基本数据类型

C语言提供了以下几种基本数据类型：
int 整型
float 浮点型
double 双精度浮点型
char 字符型

此外，C语言还定义了一些限定符，如short、long、signed、unsigned等，用于对基本数据类型进行修饰，以表示不同的数据范围。

例如，short int表示短整型，long int表示长整型，signed int表示有符号整型，unsigned int表示无符号整型。

6.3.2 复合数据类型

除了基本数据类型外，C语言还支持复合数据类型，如数组、结构体、联合体、枚举等。这些数据类型可以组合基本数据类型，形成更复杂的数据结构。

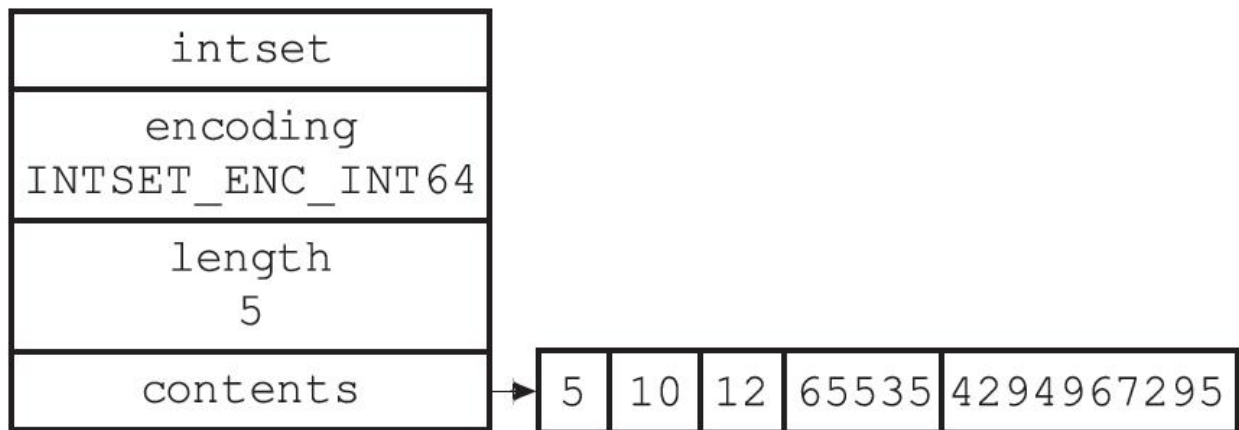
int16_t
int32_t

int16_t
int32_t
int64_t

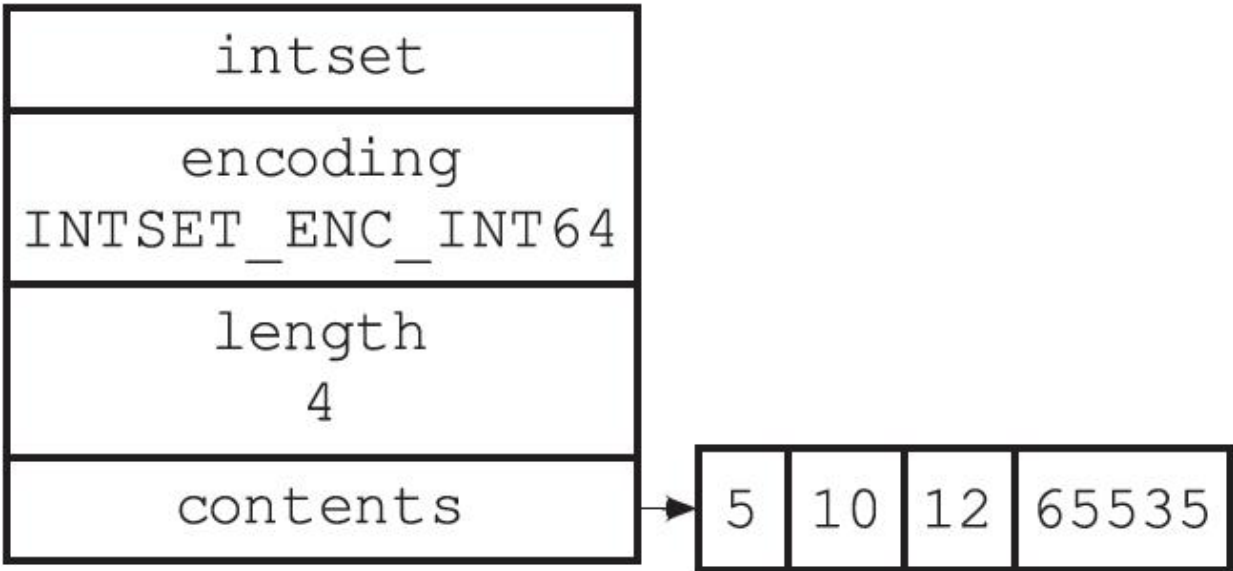
6.4 ☐☐

[illegible]

```
uint64_t i = 0;
while (i < INTSET_ENC_INT64) {
    int64_t j = 0;
```



6-11 INTSET_ENC_INT64



6-12 4 294 967 295

6.5 压缩API

图6-1 压缩API

图6-1 压缩API

函 数	作 用	时间复杂度
intsetNew	创建一个新的压缩列表	$O(1)$
intsetAdd	将给定元素添加到整数集合里面	$O(N)$
intsetRemove	从整数集合中移除给定元素	$O(N)$
intsetFind	检查给定值是否存在于集合	因为底层数组有序，查找可以通过二分查找法来进行，所以复杂度为 $O(\log N)$
intsetRandom	从整数集合中随机返回一个元素	$O(1)$
intsetGet	取出底层数组在给定索引上的元素	$O(1)$
intsetLen	返回整数集合包含的元素个数	$O(1)$
intsetBlobLen	返回整数集合占用的内存字节数	$O(1)$

6.6 □□□□

- □□□□□□□□□□□□□□

- [illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

- * □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□□□□□□□□□□□□□□

07 列表

Redis 列表使用 ziplist 作为底层实现。Redis 列表是有序的，并且可以包含字符串元素。

Redis 列表的底层实现是 ziplist。

```
redis> RPUSH lst 1 3 5 10086 "hello" "world"
(integer)6
redis> OBJECT ENCODING lst
"ziplist"
```

Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。

Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。

Redis 列表的底层实现是 ziplist。

```
redis> HMSET profile "name" "Jack" "age" 28 "job" "Programmer"
OK
redis> OBJECT ENCODING profile
"ziplist"
```

Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。Redis 列表的底层实现是 ziplist。

7.1 压缩列表

Redis 使用压缩列表（ziplist）来存储有序集合、哈希表、列表等。压缩列表是一种紧凑的二进制格式，用于存储有序集合、哈希表、列表等。它由一系列 entry 组成，每个 entry 包含一个类型标识符、一个长度、以及一个指向 entry 的指针。压缩列表的格式如下：

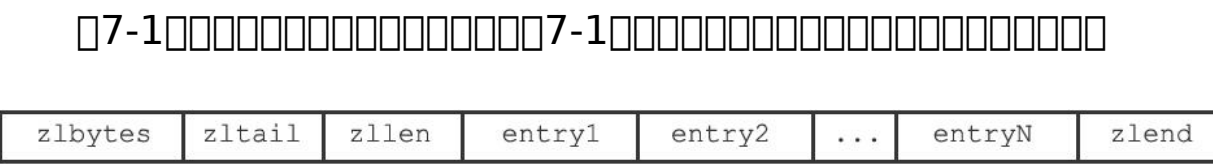


图 7-1 压缩列表的内存布局

图 7-1 压缩列表的内存布局

属性	类型	长度	用途
zlbytes	uint32_t	4 字节	记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zlend 的位置时使用
zltail	uint32_t	4 字节	记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
zllen	uint16_t	2 字节	记录了压缩列表包含的节点数量：当这个属性的值小于 UINT16_MAX（65535）时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 UINT16_MAX 时，节点的真实数量需要遍历整个压缩列表才能计算得出
entryX	列表节点	不定	压缩列表包含的各个节点，节点的长度由节点保存的内容决定
zlend	uint8_t	1 字节	特殊值 0xFF（十进制 255），用于标记压缩列表的末端

图 7-2 压缩列表的内存布局

- zlbytes 记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配，或者计算 zlend 的位置时使用
- zltail 记录压缩列表表尾节点距离压缩列表的起始地址有多少字节：通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址
- zllen 记录了压缩列表包含的节点数量：当这个属性的值小于 65535 时，这个属性的值就是压缩列表包含节点的数量；当这个值等于 65535 时，节点的真实数量需要遍历整个压缩列表才能计算得出
- entryX 压缩列表包含的各个节点，节点的长度由节点保存的内容决定
- zlend 特殊值 0xFF（十进制 255），用于标记压缩列表的末端

7.2 数据类型

数据类型

· 数据类型 $63 \times 2^6 - 1$

· 数据类型 $16383 \times 2^{14} - 1$

· 数据类型 $4294967295 \times 2^{32} - 1$

数据类型

· 4 数据类型 0 12

· 1 数据类型

· 3 数据类型

· int16_t 数据类型

· int32_t 数据类型

· int64_t 数据类型

数据类型 previous_entry_length encoding content

数据类型 7-4

previous_entry_length	encoding	content
-----------------------	----------	---------

図7-4 辞書形式の辞書

辞書形式の辞書

7.2.1 previous_entry_length

辞書のprevious_entry_lengthフィールドは、辞書の長さ（辞書の項目数）を示す。

previous_entry_lengthフィールドは1から5までの値を取る。

・previous_entry_lengthが254以下の場合は、previous_entry_lengthフィールドに1から254までの値を指定する。

・previous_entry_lengthが254以上の場合は、previous_entry_lengthフィールドに5を指定し、辞書の長さ（辞書の項目数）が254を超える場合は、0xFE（254）を指定する。

図7-5は、previous_entry_lengthフィールドが5の場合の辞書の形式を示す。

previous_entry_length 0x05	encoding ...	content ...
-------------------------------	-----------------	----------------

図7-5 辞書形式の辞書（previous_entry_length=5）

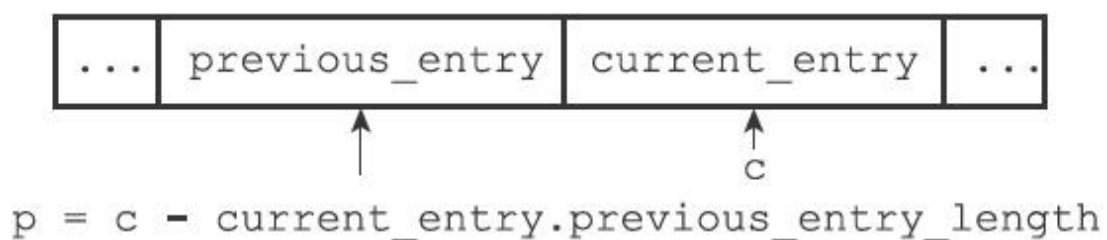
7-6 previous_entry_length
 0xFE00002766 0xFE
 previous_entry_length 0x00002766 10086
 0x00000000

previous_entry_length 0xFE00002766	encoding ...	content ...
---------------------------------------	-----------------	----------------

7-6 10086

previous_entry_length
 0x00000000

c c
 previous_entry_length p 7-7
 0x00000000



7-7

previous_entry_length
 0x00000000

7-8

· entry4 p1
zltail

· p1 entry4 previous_entry_length
entry4 entry3 p2

· p2 entry3 previous_entry_length
entry3 entry2 p3

· p3 entry2 previous_entry_length
entry2 entry1 p4 entry1

·

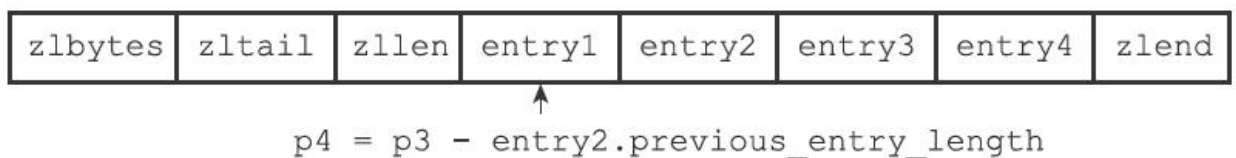
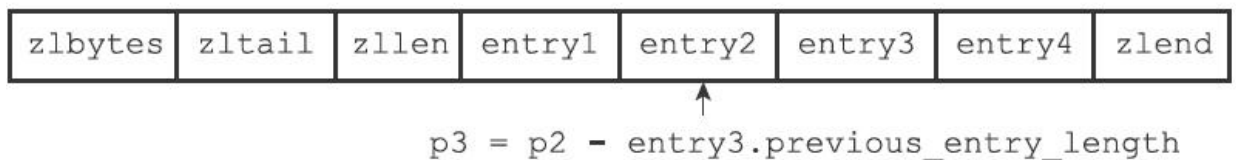
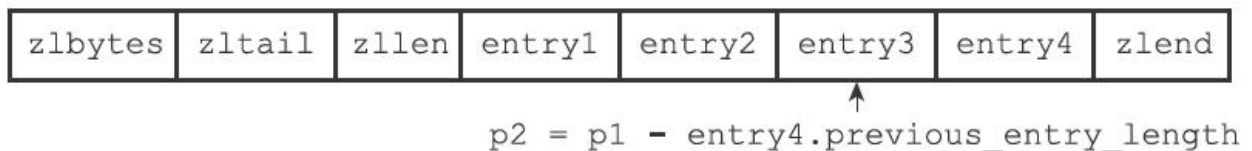
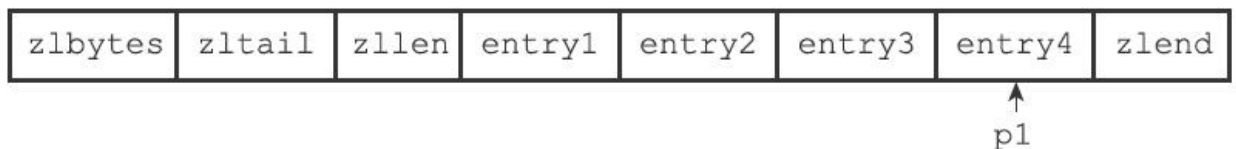


图7-8 二进制内容

7.2.2 encoding

encoding 属性保存 content 属性的值

· 00 01 10 二进制内容
content 属性保存的值

· 11 二进制内容
content 属性保存的值

图7-2 二进制内容 图7-3 二进制内容 “_”
b x 二进制内容

图7-2 二进制内容

编 码	编码长度	content 属性保存的值
00bbbbbb	1 字节	长度小于等于 63 字节的字节数组
01bbbbbb xxxxxxxx	2 字节	长度小于等于 16 383 字节的字节数组
10 _____ aaaaaaaaaa bbbbbbbb cccccccc dddddddd	5 字节	长度小于等于 4 294 967 295 的字节数组

图7-3 二进制内容

编码	编码长度	content 属性保存的值
11000000	1 字节	int16_t 类型的整数
11010000	1 字节	int32_t 类型的整数
11100000	1 字节	int64_t 类型的整数
11110000	1 字节	24 位有符号整数
11111110	1 字节	8 位有符号整数
1111xxxx	1 字节	使用这一编码的节点没有相应的 content 属性，因为编码本身的 xxxx 四个位已经保存了一个介于 0 和 12 之间的值，所以它无须 content 属性

7.2.3 content

1. content 属性保存的值

 2. encoding 属性

- 7-9 节点
- 00 节点
- 001011 节点
- content 属性 "hello world"

previous_entry_length ...	encoding 00001011	content "hello world"
------------------------------	----------------------	--------------------------

图 7-9 节点 "hello world"

7-10 节点

previous_entry_length ...	encoding 11000000	content 10086
------------------------------	----------------------	------------------

7-10 10086

·11000000int16_t

·content10086

7.3 压缩

压缩后的数据块previous_entry_length

· 压缩后的数据块previous_entry_length1
压缩后的数据块

· 压缩后的数据块previous_entry_length5
压缩后的数据块

压缩后的数据块250253
e1eN7-11

zlbytes	zltail	zllen	e1	e2	e3	...	eN	zlend
---------	--------	-------	----	----	----	-----	----	-------

7-11 压缩e1eN

e1eN254
previous_entry_lengthe1eN
previous_entry_length1

压缩后的数据块254new
e17-12



↑
添加新节点

图7-12 zlist结构

在e1的previous_entry_length后面添加1个字节，表示new的previous_entry_length为1，即5个字节。

在e1后面添加250个字节，253个字节，previous_entry_length后面添加e1的previous_entry_length为254，即257个字节，1个字节previous_entry_length为1。

在e2的previous_entry_length后面添加e1的previous_entry_length为1，即5个字节。

在e1后面添加e2的previous_entry_length为e3的previous_entry_length为e4的previous_entry_length为eN的previous_entry_length为eN。

Redis在每次更新时，都会调用“cascade update”函数，图7-13展示了其工作原理。

在每次更新时，都会调用“cascade update”函数，图7-13展示了其工作原理。

图7-14展示了e1到eN的扩展过程。当e1的previous_entry_length属性为254时，表示其指向的entry已经扩展到了253。当e2的previous_entry_length属性为254时，表示其指向的entry已经扩展到了253。当eN的previous_entry_length属性为254时，表示其指向的entry已经扩展到了253。当e1的previous_entry_length属性为254时，表示其指向的entry已经扩展到了253。当e1的previous_entry_length属性为254时，表示其指向的entry已经扩展到了253。

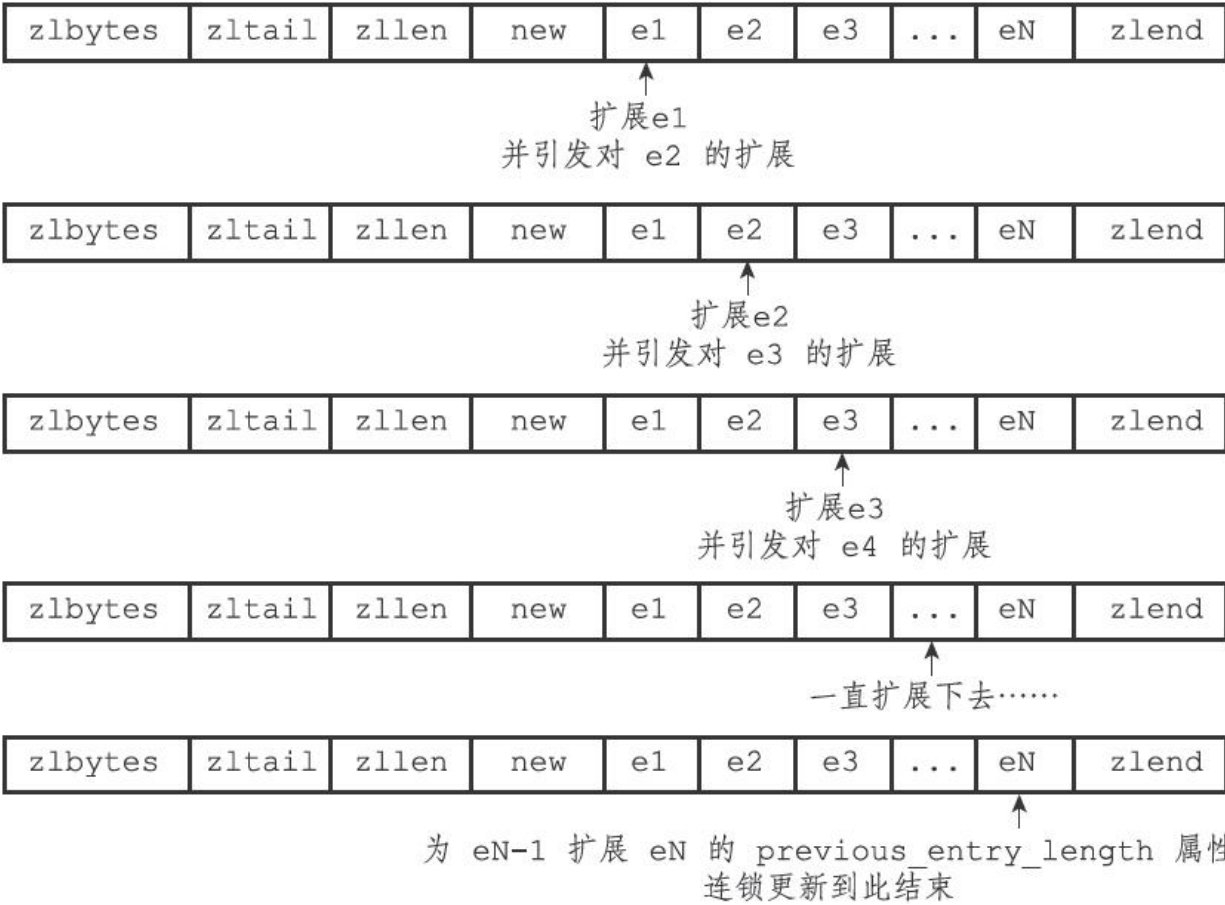


图7-13 删除small节点



图7-14 扩展e1到eN

$\text{N} \rightarrow \text{O}$
 $\text{N} \rightarrow \text{O} \text{N}^2$

[illegible]

• 250 253

[illegible]

ziplistPush $O(N)$

7.4 压缩列表API

图7-4 压缩列表API

图7-4 压缩列表API

函数	作用	算法复杂度
ziplistNew	创建一个新的压缩列表	$O(1)$
ziplistPush	创建一个包含给定值的新节点，并将这个新节点添加到压缩列表的表头或者表尾	平均 $O(N)$ ，最坏 $O(N^2)$
ziplistInsert	将包含给定值的新节点插入到给定节点之后	平均 $O(N)$ ，最坏 $O(N^2)$
ziplistIndex	返回压缩列表给定索引上的节点	$O(N)$
ziplistFind	在压缩列表中查找并返回包含了给定值的节点	因为节点的值可能是一个字节数组，所以检查节点值和给定值是否相同的复杂度为 $O(N)$ ，而查找整个列表的复杂度则为 $O(N^2)$
ziplistNext	返回给定节点的下一个节点	$O(1)$
ziplistPrev	返回给定节点的前一个节点	$O(1)$
ziplistGet	获取给定节点所保存的值	$O(1)$
ziplistDelete	从压缩列表中删除给定的节点	平均 $O(N)$ ，最坏 $O(N^2)$
ziplistDeleteRange	删除压缩列表在给定索引上的连续多个节点	平均 $O(N)$ ，最坏 $O(N^2)$
ziplistBlobLen	返回压缩列表目前占用的内存字节数	$O(1)$
ziplistLen	返回压缩列表目前包含的节点数量	节点数量小于 65 535 时为 $O(1)$ ，大于 65 535 时为 $O(N)$

ziplistPush、ziplistInsert、ziplistDelete

ziplistDeleteRange 的时间复杂度为 $O(N^2)$

7.5 四角

- [illegible]

111

8 8

Redis Redis
SDS

Redis
Redis
Redis

Redis Redis
Redis
Redis

Redis Redis
Redis Redis
Redis

Redis
maxmemory

Redis

8.1 Redis 入门

Redis 是一个开源的分布式键值数据库。Redis 支持多种数据类型，如字符串、哈希、列表、集合、有序集合、位图等。Redis 还支持持久化、主从复制、集群等功能。

Redis 提供了 SET 命令，用于设置键值对。例如，设置键 msg 的值为 hello world。

```
"msg" SET "hello world"
```

```
redis> SET msg "hello world"
OK
```

Redis 使用 redisObject 结构体来存储数据。该结构体包含以下成员：

```
type 数据类型
encoding 编码
ptr 指向数据的指针
```

```
typedef struct redisObject {
    //
    unsigned type:4;
    //
    unsigned encoding:4;
    //
    void *ptr;
    // ...
} robj;
```

8.1.1 Redis 安装

type命令返回的对象的类型8-1

8-1

类型常量	对象的名称
REDIS_STRING	字符串对象
REDIS_LIST	列表对象
REDIS_HASH	哈希对象
REDIS_SET	集合对象
REDIS_ZSET	有序集合对象

Redis返回的对象的类型

- 字符串对象
- 列表对象

TYPE命令返回的对象的类型

```
#
redis> SET msg "hello world"
OK
redis> TYPE msg
string
```

```

#
redis> RPUSH numbers 1 3 5
(integer) 6
redis> TYPE numbers
list
#
redis> HMSET profile name Tom age 25 career Programmer
OK
redis> TYPE profile
hash
#
redis> SADD fruits apple banana cherry
(integer) 3
redis> TYPE fruits
set
#
redis> ZADD price 8.5 apple 5.0 banana 6.0 cherry
(integer) 3
redis> TYPE price
zset

```

8-2 TYPE 命令

8-2 TYPE 命令

对象	对象 type 属性的值	TYPE 命令的输出
字符串对象	REDIS_STRING	"string"
列表对象	REDIS_LIST	"list"
哈希对象	REDIS_HASH	"hash"
集合对象	REDIS_SET	"set"
有序集合对象	REDIS_ZSET	"zset"

8.1.2 字符串

ptr指向的字符串的encoding

encoding

图8-3

图8-3

编码常量	编码所对应的底层数据结构
REDIS_ENCODING_INT	long 类型的整数
REDIS_ENCODING_EMBSTR	embstr 编码的简单动态字符串
REDIS_ENCODING_RAW	简单动态字符串
REDIS_ENCODING_HT	字典
REDIS_ENCODING_LINKEDLIST	双端链表
REDIS_ENCODING_ZIPLIST	压缩列表
REDIS_ENCODING_INTSET	整数集合
REDIS_ENCODING_SKIPLIST	跳跃表和字典

图8-4

图8-4

类 型	编 码	对 象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象

OBJECT ENCODING

```
redis> SET msg "hello wrold"
OK
redis> OBJECT ENCODING msg
"embstr"
redis> SET story "long long long long long long ago ..."
OK
redis> OBJECT ENCODING story
"raw"
redis> SADD numbers 1 3 5
(integer) 3
redis> OBJECT ENCODING numbers
"intset"
redis> SADD numbers "seven"
(integer) 1
redis> OBJECT ENCODING numbers
"hashtable"
```

8-5 OBJECT ENCODING

8-5 OBJECT ENCODING

对象所使用的底层数据结构	编码常量	<i>OBJECT ENCODING</i> 命令输出
整数	REDIS_ENCODING_INT	"int"
embstr 编码的简单动态字符串 (SDS)	REDIS_ENCODING_EMBSTR	"embstr"
简单动态字符串	REDIS_ENCODING_RAW	"raw"
字典	REDIS_ENCODING_HT	"hashtable"

(续)

对象所使用的底层数据结构	编码常量	<i>OBJECT ENCODING</i> 命令输出
双端链表	REDIS_ENCODING_LINKEDLIST	"linkedlist"
压缩列表	REDIS_ENCODING_ZIPLIST	"ziplist"
整数集合	REDIS_ENCODING_INTSET	"intset"
跳跃表和字典	REDIS_ENCODING_SKIPLIST	"skiplist"

1. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

2. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

3. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

4. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

5. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

6. 使用 `encoding` 命令可以查看 Redis 对象所使用的底层数据结构。

8.2 数据类型

redisObject *raw, embstr

long

ptr void* long int

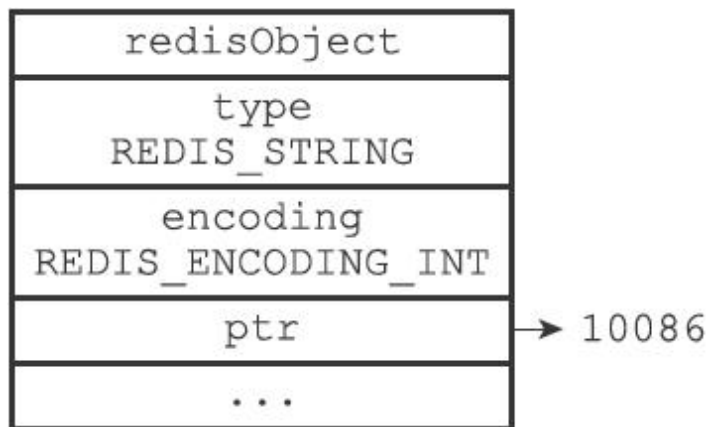


图8-1 int数据类型

SET 8-1 int

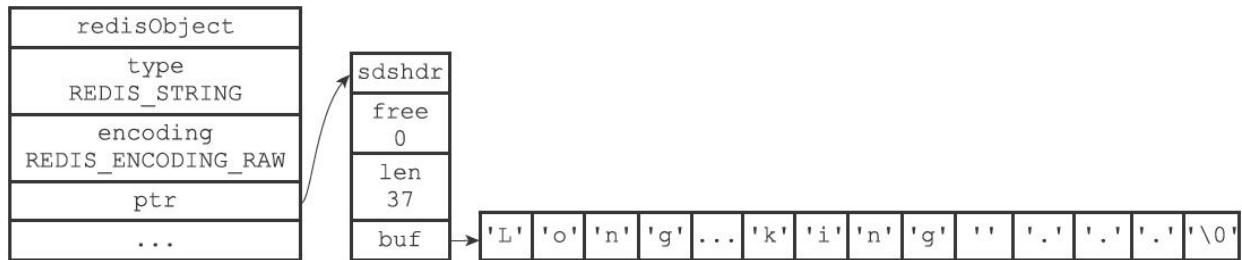
number

```
redis> SET number 10086
OK
redis> OBJECT ENCODING number
"int"
```

32

SDS raw

redisObject 8-2 raw
story



8-2 raw

```
redis> SET story "Long, long ago there lived a king ..."
OK
redis> STRLEN story
(integer) 37
redis> OBJECT ENCODING story
"raw"
```

redisObject 32 embstr

embstr raw
redisObject sdshdr raw
redisObject sdshdr embstr
redisObject sdshdr 8-3



8-3 embstr

redis> SET pi 3.14

OK

redis> OBJECT ENCODING pi
"embstr"

redis> SET pi 3.14

redis> INCRBYFLOAT pi 2.0

"5.14"
redis> OBJECT ENCODING pi
"embstr"

redis> SET pi 3.14

8-6 数据类型

值	编码
可以用 long 类型保存的整数	int
可以用 long double 类型保存的浮点数	embstr 或者 raw
字符串值，或者因为长度太大而没办法用 long 类型表示的整数，又或者因为长度太大而没办法用 long double 类型表示的浮点数	embstr 或者 raw

8.2.1 数据类型

int embstr raw
int

int
int raw

APPEND
10086 "10086"
raw

```
redis> SET number 10086
OK
redis> OBJECT ENCODING number
"int"
redis> APPEND number " is a good number!"
(integer) 23
redis> GET number
"10086 is a good number!"
redis> OBJECT ENCODING number
"raw"
```

Redis embstr int
raw embstr
embstr raw
embstr raw
embstr raw

embstr APPEND

embstr raw

```
redis> SET msg "hello world"
OK
redis> OBJECT ENCODING msg
"embstr"
redis> APPEND msg " again!"
(integer) 18
redis> OBJECT ENCODING msg
"raw"
```

8.2.2

8-7

8-7

命令	int 编码的实现方法	embstr 编码的实现方法	raw 编码的实现方法
<i>SET</i>	使用 int 编码保存值	使用 embstr 编码保存值	使用 raw 编码保存值
<i>GET</i>	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后向客户端返回这个字符串值	直接向客户端返回字符串值	直接向客户端返回字符串值
<i>APPEND</i>	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	将对象转换成 raw 编码，然后按 raw 编码的方式执行此操作	调用 sdscatlen 函数，将给定字符串追加到现有字符串的末尾
<i>INCRBYFLOAT</i>	取出整数值并将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误	取出字符串值并尝试将其转换成 long double 类型的浮点数，对这个浮点数进行加法计算，然后将得出的浮点数结果保存起来。如果字符串值不能被转换成浮点数，那么向客户端返回一个错误
<i>INCRBY</i>	对整数值进行加法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
<i>DECRBY</i>	对整数值进行减法计算，得出的计算结果会作为整数被保存起来	embstr 编码不能执行此命令，向客户端返回一个错误	raw 编码不能执行此命令，向客户端返回一个错误
<i>STRLEN</i>	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，计算并返回这个字符串值的长度	调用 sdslen 函数，返回字符串的长度	调用 sdslen 函数，返回字符串的长度
<i>SETRANGE</i>	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将对象转换成 raw 编码，然后按 raw 编码的方式执行此命令	将字符串特定索引上的值设置为给定的字符
<i>GETRANGE</i>	拷贝对象所保存的整数值，将这个拷贝转换成字符串值，然后取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符	直接取出并返回字符串指定索引上的字符

8.3 内部实现

Redis 使用 `ziplist` 和 `linkedlist`

`ziplist` 是 Redis 内部使用的一种紧凑的二进制数据结构，它由一系列的 `entry` 组成。

Redis 使用 `RPUSH` 命令向列表的尾部添加元素。例如，向 `numbers` 列表添加元素：

```
redis> RPUSH numbers 1 "three" 5
(integer) 3
```

此时，`numbers` 列表在内存中的结构如下所示（图 8-5）。

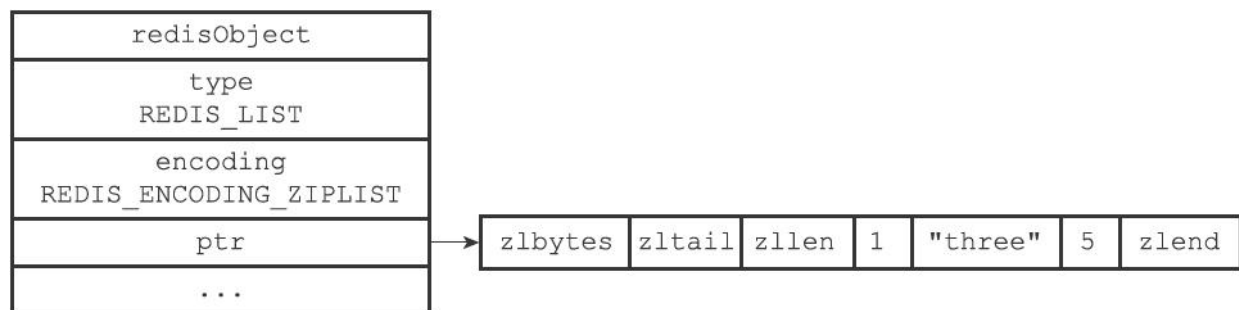


图 8-5 `ziplist` 列表的内存结构

Redis 使用 `linkedlist` 来管理列表中的元素。每个元素都是一个 `node` 结构。

Redis 使用 `list` 命令来操作列表。例如，向列表添加元素：

Redis 使用 `numbers` 列表来存储数字。此时，`numbers` 列表在内存中的结构如下所示（图 8-6）。

Redis 使用 `linkedlist` 来管理列表中的元素。每个元素都是一个 `node` 结构。

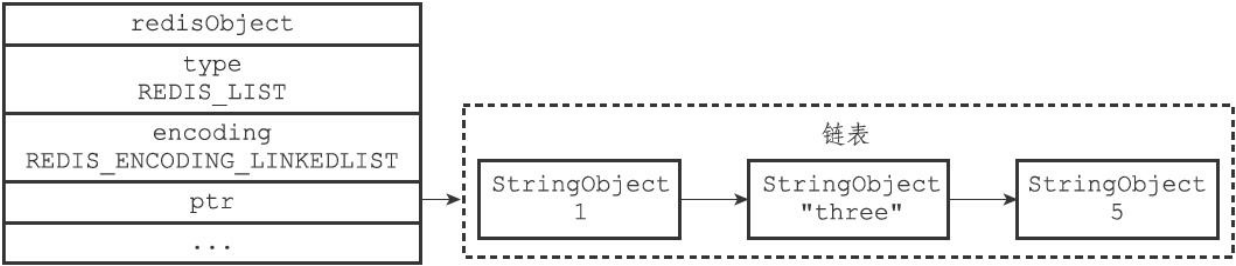


图8-6 linkedlist中的numbers

在linkedlist中，每个节点都是一个StringObject，它指向下一个节点。Redis使用这种结构来存储列表。每个节点都是一个StringObject，它指向下一个节点。



在图8-6中，每个StringObject都指向下一个StringObject。图8-7显示了StringObject的结构。图8-8显示了StringObject的内存布局。

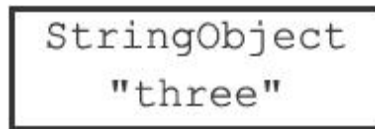


图8-7 StringObject结构

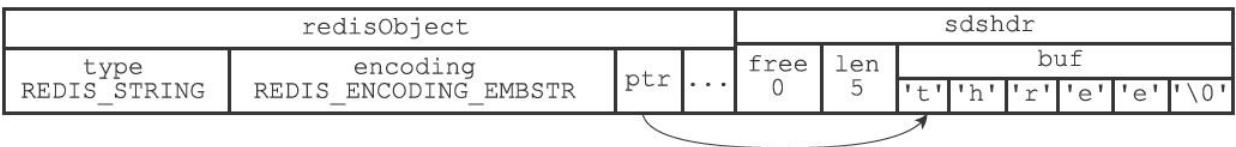


图8-8 StringObject内存布局

在图8-8中，每个StringObject都指向下一个StringObject。

8.3.1 配置

Redis 默认使用 ziplist 存储

· 默认使用 64 字节

· 默认使用 512 字节存储 linkedlist



Redis 配置文件中 list-max-ziplist-value

list-max-ziplist-entries

ziplist 存储的 ziplist 大小

ziplist 存储的 ziplist 数量

ziplist 存储 linkedlist

Redis 配置文件中

```
#
64
redis> RPUSH blah "hello" "world" "again"
(integer)3
redis> OBJECT ENCODING blah
"ziplist"
#
65
redis> RPUSH blah
"www"
(integer) 4
```

```
#
redis> OBJECT ENCODING blah
"linkedlist"
```

redis> OBJECT ENCODING integers

```
#
redis> EVAL "for i=1, 512 do redis.call('RPUSH', KEYS[1],i)end" 1 "integers"
(nil)
redis> LLEN integers
(integer) 512
redis> OBJECT ENCODING integers
"ziplist"
#
redis> RPUSH integers 513
(integer) 513
#
redis> OBJECT ENCODING integers
"linkedlist"
```

8.3.2 8-8

redis> OBJECT ENCODING integers

redis> OBJECT ENCODING integers

8-8

命令	ziplist 编码的实现方法	linkedList 编码的实现方法
<i>LPUSH</i>	调用 <code>ziplistPush</code> 函数，将新元素推入到压缩列表的表头	调用 <code>listAddNodeHead</code> 函数，将新元素推入到双端链表的表头
<i>RPUSH</i>	调用 <code>ziplistPush</code> 函数，将新元素推入到压缩列表的表尾	调用 <code>listAddNodeTail</code> 函数，将新元素推入到双端链表的表尾
<i>LPOP</i>	调用 <code>ziplistIndex</code> 函数定位压缩列表的表头节点，在向用户返回节点所保存的元素之后，调用 <code>ziplistDelete</code> 函数删除表头节点	调用 <code>listFirst</code> 函数定位双端链表的表头节点，在向用户返回节点所保存的元素之后，调用 <code>listDelNode</code> 函数删除表头节点
<i>RPOP</i>	调用 <code>ziplistIndex</code> 函数定位压缩列表的表尾节点，在向用户返回节点所保存的元素之后，调用 <code>ziplistDelete</code> 函数删除表尾节点	调用 <code>listLast</code> 函数定位双端链表的表尾节点，在向用户返回节点所保存的元素之后，调用 <code>listDelNode</code> 函数删除表尾节点
<i>LINDEX</i>	调用 <code>ziplistIndex</code> 函数定位压缩列表中的指定节点，然后返回节点所保存的元素	调用 <code>listIndex</code> 函数定位双端链表中的指定节点，然后返回节点所保存的元素
<i>LLEN</i>	调用 <code>ziplistLen</code> 函数返回压缩列表的长度	调用 <code>listLength</code> 函数返回双端链表的长度
<i>LINSERT</i>	插入新节点到压缩列表的表头或者表尾时，使用 <code>ziplistPush</code> 函数；插入新节点到压缩列表的其他位置时，使用 <code>ziplistInsert</code> 函数	调用 <code>listInsertNode</code> 函数，将新节点插入到双端链表的指定位置
<i>LREM</i>	遍历压缩列表节点，并调用 <code>ziplistDelete</code> 函数删除包含了给定元素的节点	遍历双端链表节点，并调用 <code>listDelNode</code> 函数删除包含了给定元素的节点
<i>LTRIM</i>	调用 <code>ziplistDeleteRange</code> 函数，删除压缩列表中所有不在指定索引范围内的节点	遍历双端链表节点，并调用 <code>listDelNode</code> 函数删除链表中所有不在指定索引范围内的节点
<i>LSET</i>	调用 <code>ziplistDelete</code> 函数，先删除压缩列表指定索引上的现有节点，然后调用 <code>ziplistInsert</code> 函数，将一个包含给定元素的新节点插入到相同索引上面	调用 <code>listIndex</code> 函数，定位到双端链表指定索引上的节点，然后通过赋值操作更新节点的值

8.4 字典

字典由 ziplist 和 hashtable 组成

ziplist 是 Redis 2.8 版本引入的新的数据结构，它是由一系列 ziplist 元素组成的。每个 ziplist 元素都是一个字符串，这个字符串的格式如下：

- 字符串的开头是一个字节，表示字符串的长度（以字节为单位）

- 字符串的后面是字符串的内容

字典的格式如下：

字典由 HSET 命令创建，每个字典都有一个 profile 名称

```
redis> HSET profile name "Tom"
(integer) 1
redis> HSET profile age 25
(integer) 1
redis> HSET profile career "Programmer"
(integer) 1
```

字典的 profile 名称和 ziplist 元素在 Redis 8-9 版本中都有使用，在 Redis 8-10 版本中也有使用。

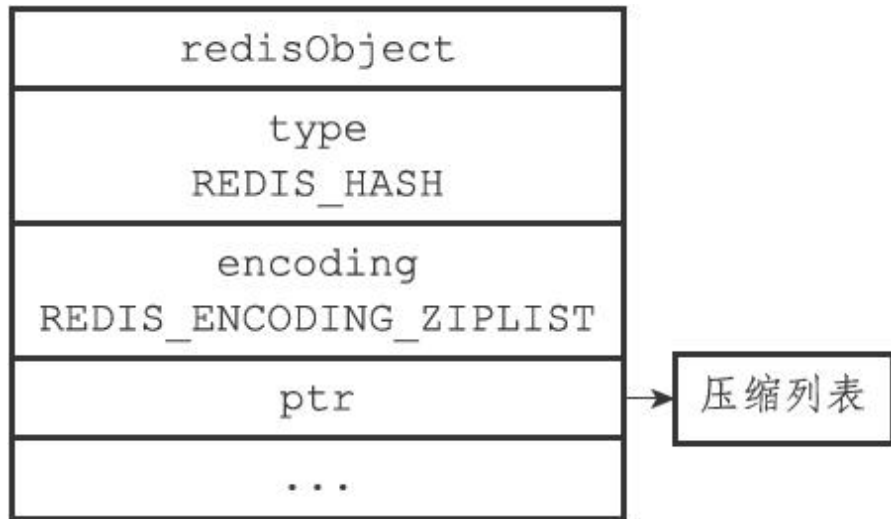


图8-9 ziplist内部profile结构

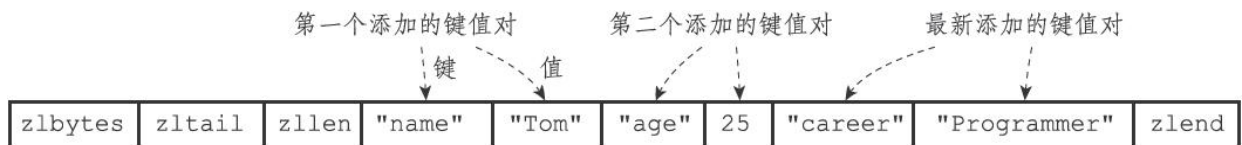


图8-10 profile内部结构

Redis使用hashtable来存储键值对，每个键值对都是一个ziplist结构，每个ziplist结构都是一个压缩列表。压缩列表是一个有序列表，每个元素都是一个键值对。压缩列表的每个元素都是一个键值对，键和值都是字符串。压缩列表的每个元素都是一个键值对，键和值都是字符串。压缩列表的每个元素都是一个键值对，键和值都是字符串。

· 压缩列表的每个元素都是一个键值对

· 压缩列表的每个元素都是一个键值对

Redis使用profile来记录ziplist的统计信息，每个ziplist都有一个profile。profile是一个结构体，包含了一些统计信息。profile的结构如下：

图8-11 profile结构

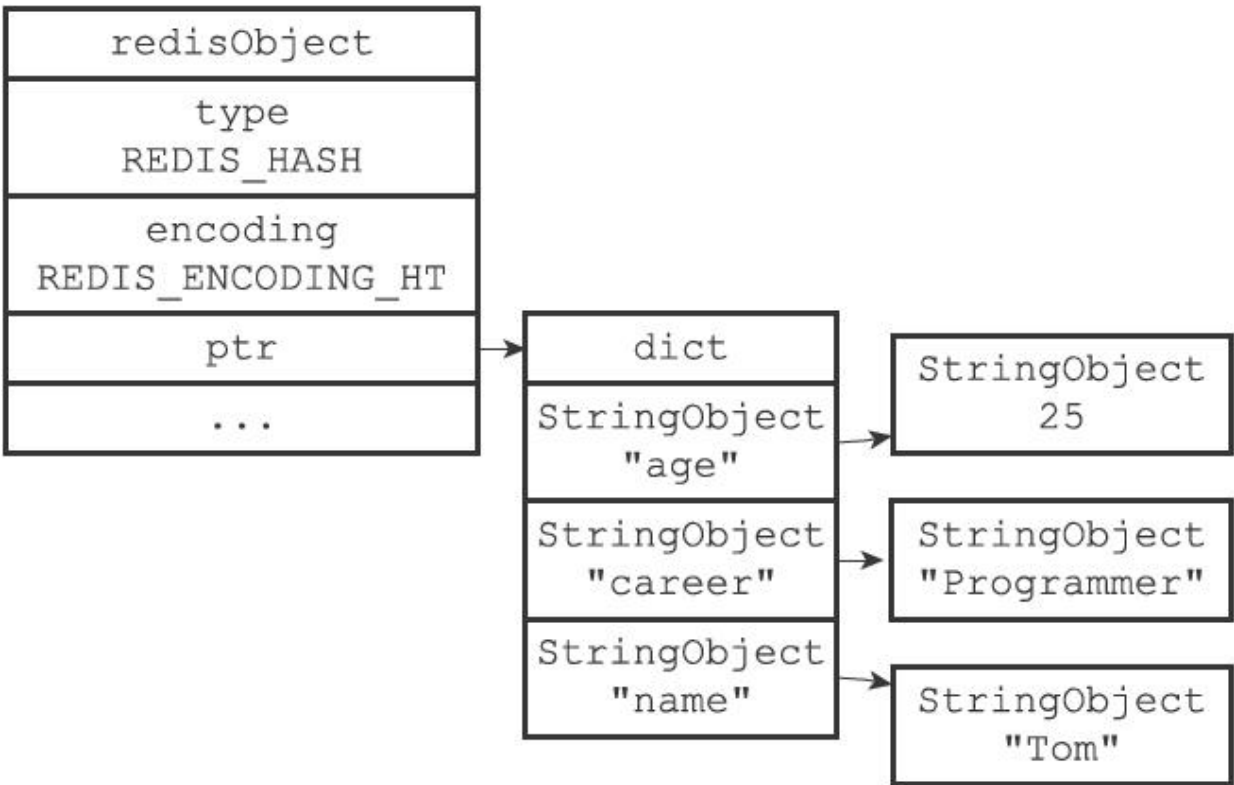


图8-11 hashtable的profile

8.4.1 配置

Redis的配置文件位于redis.conf，其中配置了以下参数：

- `hash-max-ziplist-entries`：哈希表的最大ziplist条目数，默认为64。

- `hash-max-ziplist-value`：哈希表的最大ziplist值大小，默认为512字节。

图



hash-max-ziplist-value
hash-max-ziplist-entries

ziplist
ziplist
ziplist
hashtable

```
#
64
redis> HSET book name "Mastering C++ in 21 days"
(integer) 1
redis> OBJECT ENCODING book
"ziplist"
#
66
redis> HSET book
long_long_long_long_long_long_long_long_long_long_description
"content"
(integer) 1
#
redis> OBJECT ENCODING book
"hashtable"
```

```
#
64
redis> HSET blah greeting "hello world"
```



```

(integer) 1
redis> OBJECT ENCODING blah
"ziplist"
#
████████████████████████████████████████68
██
redis> HSET blah story "many string ... many string ... many string ... many
string ... many"
(integer) 1
#
██████
redis> OBJECT ENCODING blah
"hashtable"

```

```

████████████████████████████████████████████████████████████████████████████████

```

```

#
████████512
██████████████
redis> EVAL "for i=1, 512 do redis.call('HSET', KEYS[1], i, i)end" 1 "numbers"
(nil)
redis> HLEN numbers
(integer) 512
redis> OBJECT ENCODING numbers
"ziplist"
#
████████████████████████████████████████████████████████████████████████████████513
█
redis> HMSET numbers "key" "value"
OK
redis> HLEN numbers
(integer) 513
#
██████
redis> OBJECT ENCODING numbers
"hashtable"

```

8.4.2 ██████████

8-9

8-9

命令	ziplist 编码实现方法	hashtable 编码的实现方法
HSET	首先调用 ziplistPush 函数，将键推入到压缩列表的表尾，然后再次调用 ziplistPush 函数，将值推入到压缩列表的表尾	调用 dictAdd 函数，将新节点添加到字典里面
HGET	首先调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后调用 ziplistNext 函数，将指针移动到键节点旁边的值节点，最后返回值节点	调用 dictFind 函数，在字典中查找给定键，然后调用 dictGetVal 函数，返回该键所对应的值
HEXISTS	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，如果找到的话说明键值对存在，没找到的话就说明键值对不存在	调用 dictFind 函数，在字典中查找给定键，如果找到的话说明键值对存在，没找到的话就说明键值对不存在
HDEL	调用 ziplistFind 函数，在压缩列表中查找指定键所对应的节点，然后将相应的键节点、以及键节点旁边的值节点都删除掉	调用 dictDelete 函数，将指定键所对应的键值对从字典中删除掉
HLEN	调用 ziplistLen 函数，取得压缩列表包含节点的总数量，将这个数量除以 2，得出的结果就是压缩列表保存的键值对的数量	调用 dictSize 函数，返回字典包含的键值对数量，这个数量就是哈希对象包含的键值对数量
HGETALL	遍历整个压缩列表，用 ziplistGet 函数返回所有键和值（都是节点）	遍历整个字典，用 dictGetKey 函数返回字典的键，用 dictGetVal 函数返回字典的值

8.5 集合

集合的底层实现是 `intset` 和 `hashtable`

`intset` 是一个有序数组，用于存储整数

图 8-12 展示了 `intset` 的结构

```
redis> SADD numbers 1 3 5
(integer) 3
```

集合的底层实现是 `hashtable`，用于存储字符串

如果集合中不包含任何元素，则 `ptr` 指向 `NULL`

图 8-13 展示了 `hashtable` 的结构

```
redis> SADD Dfruits "apple" "banana" "cherry"
(integer) 3
```

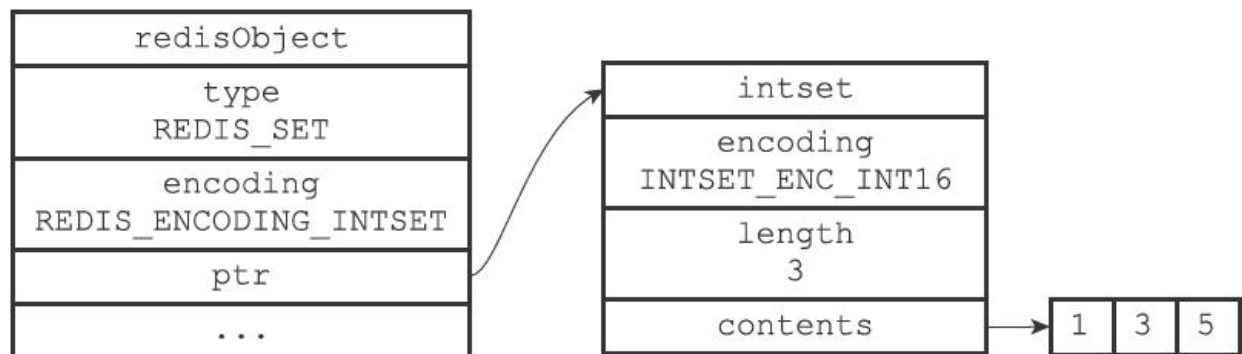


图 8-12 `intset` 结构

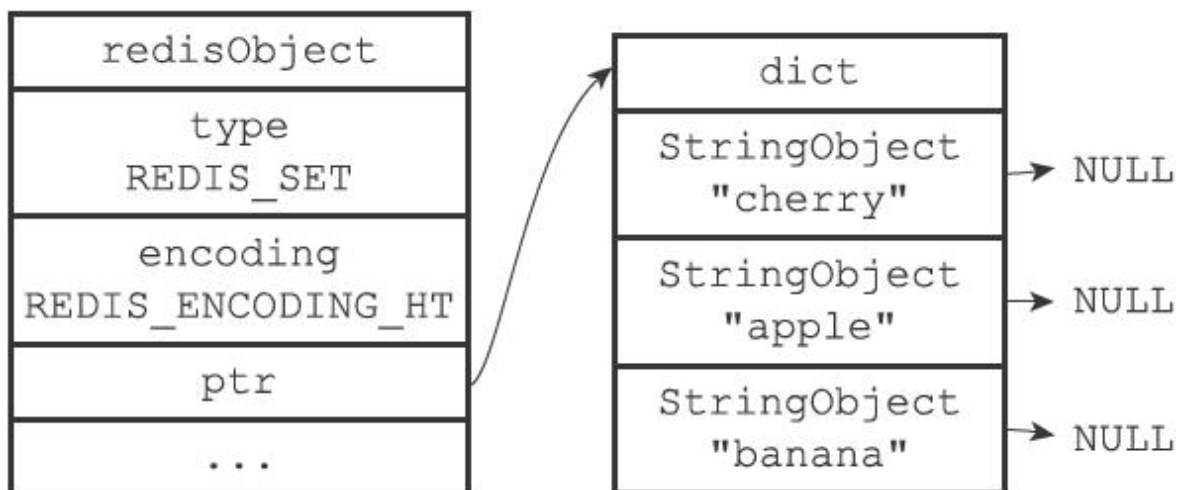


图8-13 hashtable的fruits

8.5.1 初始化

初始化intset结构

· 初始化intset结构

· 初始化intset结构的512个元素

初始化hashtable结构



初始化set-max-intset-entries

初始化

初始化intset结构

初始化intset结构

intset hashtable

intset

```
redis> SADD numbers 1 3 5
(integer) 3
redis> OBJECT ENCODING numbers
"intset"
```

```
redis> SADD numbers "seven"
(integer) 1
redis> OBJECT ENCODING numbers
"hashtable"
```

512 intset

```
redis> EVAL "for i=1, 512 do redis.call('SADD', KEYS[1], i) end" 1 integers
(nil)
redis> SCARD integers
(integer) 512
redis> OBJECT ENCODING integers
"intset"
```

513

```
redis> SADD integers 10086
(integer) 1
```

```
redis> SCARD integers
(integer) 513
redis> OBJECT ENCODING integers
"hashtable"
```

8.5.2 集合的编码

Redis 的集合对象使用 intset 或 hashtable 编码。intset 编码适用于集合元素数量在 8-10 个且元素值在 32 位有符号整数范围内的情况。hashtable 编码适用于集合元素数量超过 10 个或元素值超出 32 位有符号整数范围的情况。

命令	intset 编码的实现方法	hashtable 编码的实现方法
SADD	调用 intsetAdd 函数，将所有新元素添加到整数集合里面	调用 dictAdd，以新元素为键，NULL 为值，将键值对添加到字典里面
(续)		
命令	intset 编码的实现方法	hashtable 编码的实现方法
SCARD	调用 intsetLen 函数，返回整数集合所包含的元素数量，这个数量就是集合对象所包含的元素数量	调用 dictSize 函数，返回字典所包含的键值对数量，这个数量就是集合对象所包含的元素数量
SISMEMBER	调用 intsetFind 函数，在整数集合中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合	调用 dictFind 函数，在字典的键中查找给定的元素，如果找到了说明元素存在于集合，没找到则说明元素不存在于集合
SMEMBERS	遍历整个整数集合，使用 intsetGet 函数返回集合元素	遍历整个字典，使用 dictGetKey 函数返回字典的键作为集合元素
SRANDMEMBER	调用 intsetRandom 函数，从整数集合中随机返回一个元素	调用 dictGetRandomKey 函数，从字典中随机返回一个字典键
SPOP	调用 intsetRandom 函数，从整数集合中随机取出一个元素，在将这个随机元素返回给客户端之后，调用 intsetRemove 函数，将随机元素从整数集合中删除掉	调用 dictGetRandomKey 函数，从字典中随机取出一个字典键，在将这个随机字典键的值返回给客户端之后，调用 dictDelete 函数，从字典中删除随机字典键所对应的键值对
SREM	调用 intsetRemove 函数，从整数集合中删除所有给定的元素	调用 dictDelete 函数，从字典中删除所有键为给定元素的键值对

8.6 有序集合

有序集合使用ziplist和skiplist

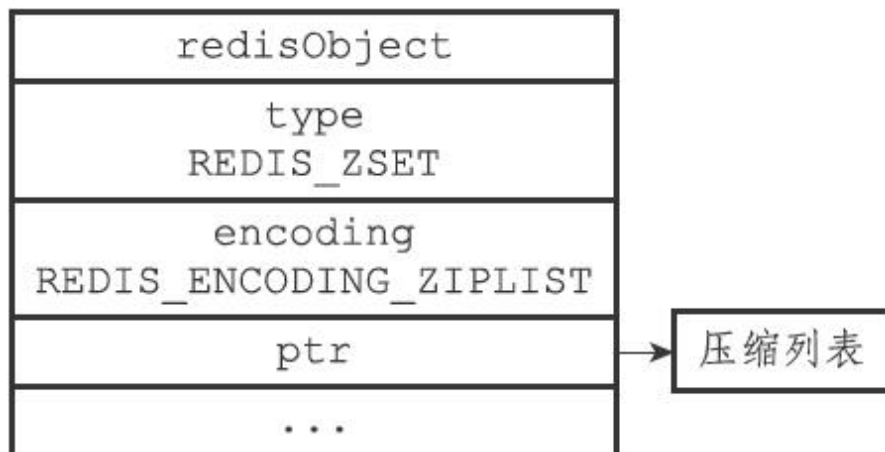
ziplist使用紧凑的二进制格式存储成员和分数。成员和分数以二进制形式存储在ziplist中。成员和分数的存储顺序与它们在集合中的顺序相同。成员和分数的存储顺序与它们在集合中的顺序相同。

有序集合使用skiplist来存储成员和分数。skiplist是一种概率平衡的链表，它允许快速查找成员和分数。skiplist的每个节点包含一个成员、一个分数、以及指向下一个节点的指针。skiplist的每个节点还包含一个指向下一个节点的指针，该指针指向下一个节点的下一个节点。

有序集合使用ZADD命令来添加成员和分数。ZADD命令的语法如下：

```
redis> ZADD price 8.5 apple 5.0 banana 6.0 cherry  
(integer) 3
```

price有序集合使用ziplist来存储成员和分数。8-14展示了price有序集合的ziplist结构。8-15展示了price有序集合的skiplist结构。



```
skiplist[]zset[]zset[]
[]
```

The diagram illustrates a linked list structure. It consists of a horizontal sequence of nodes, each represented by a box divided into two parts: a string field and a float field. The nodes are labeled as follows:

- Node 1:** String field contains "banana", Float field contains 5.0. Above this node, the text "分值最少的元素" (Element with the lowest score) has an arrow pointing to the string field, and "成员" (Member) has an arrow pointing to the float field.
- Node 2:** String field contains "cherry", Float field contains 6.0. Above this node, the text "分值排第二的元素" (Element with the second lowest score) has an arrow pointing to the string field, and "分值" (Score) has an arrow pointing to the float field.
- Node 3:** String field contains "apple", Float field contains 8.5. Above this node, the text "分值最大的元素" (Element with the highest score) has an arrow pointing to the string field.

The sequence of nodes is preceded by "zlbytes", "zltail", and "zllen", and followed by "zlend".

```
zset zsl
object score
ZRANK ZRANGE API
```

```

double
zset

```


Redis 的有序集合（Sorted Set）是 Redis 提供的一种数据类型，它允许将元素与一个浮点数值（score）关联起来，并根据这个 score 对元素进行排序。有序集合的底层实现是跳表（skiplist）和压缩列表（ziplist）。

有序集合的底层实现

有序集合的底层实现主要依赖于跳表（skiplist）和压缩列表（ziplist）。跳表是一种概率数据结构，它允许在 $O(\log N)$ 的时间复杂度内查找、插入和删除元素。在 Redis 中，跳表的每个节点包含一个或多个指向下一个节点的指针，这些指针的个数是根据节点的 score 值随机决定的。跳表的查找过程类似于二分查找，通过不断比较当前节点的 score 值与目标值，直到找到目标元素或到达表的末尾。跳表的插入和删除操作也需要遍历跳表，找到合适的位置进行插入或删除。跳表的查找、插入和删除操作的时间复杂度都是 $O(\log N)$ 。跳表的查找、插入和删除操作的时间复杂度都是 $O(\log N)$ 。

除了跳表，Redis 还使用压缩列表（ziplist）来存储有序集合。压缩列表是一种紧凑的数据结构，它允许将多个元素存储在一个连续的内存块中。在 Redis 中，压缩列表的每个元素包含一个浮点数值（score）和一个指向元素的指针。压缩列表的查找、插入和删除操作的时间复杂度都是 $O(N)$ 。压缩列表的查找、插入和删除操作的时间复杂度都是 $O(N)$ 。

Redis 的有序集合使用跳表（skiplist）和压缩列表（ziplist）来存储元素。跳表的查找、插入和删除操作的时间复杂度都是 $O(\log N)$ 。压缩列表的查找、插入和删除操作的时间复杂度都是 $O(N)$ 。Redis 的有序集合使用跳表（skiplist）和压缩列表（ziplist）来存储元素。跳表的查找、插入和删除操作的时间复杂度都是 $O(\log N)$ 。压缩列表的查找、插入和删除操作的时间复杂度都是 $O(N)$ 。


```

(nil)
redis> ZCARD numbers
(integer) 128
redis> OBJECT ENCODING numbers
"ziplist"
#
██████████
redis> ZADD numbers 3.14 pi
(integer) 1
#
██████████████████129
█
redis> ZCARD numbers
(integer) 129
#
██████
redis> OBJECT ENCODING numbers
"skiplist"

```

██

```

#
██████████████████████████████████████
redis> ZADD blah 1.0 www
(integer) 1
redis> OBJECT ENCODING blah
"ziplist"
#
██████████████████66
██████
redis> ZADD blah 2.0
oooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
(integer) 1
#
██████
redis> OBJECT ENCODING blah
"skiplist"

```

8.6.2 ██████████

8-11

8-11

命令	ziplist 编码的实现方法	zset 编码的实现方法
ZADD	调用 ziplistInsert 函数，将成员和分值作为两个节点分别插入到压缩列表	先调用 zslInsert 函数，将新元素添加到跳跃表，然后调用 dictAdd 函数，将新元素关联到字典
ZCARD	调用 ziplistLen 函数，获得压缩列表包含节点的数量，将这个数量除以 2 得出集合元素的数量	访问跳跃表数据结构的 length 属性，直接返回集合元素的数量
ZCOUNT	遍历压缩列表，统计分值在给定范围内的节点的数量	遍历跳跃表，统计分值在给定范围内的节点的数量
ZRANGE	从表头向表尾遍历压缩列表，返回给定索引范围内的所有元素	从表头向表尾遍历跳跃表，返回给定索引范围内的所有元素
ZREVRANGE	从表尾向表头遍历压缩列表，返回给定索引范围内的所有元素	从表尾向表头遍历跳跃表，返回给定索引范围内的所有元素
ZRANK	从表头向表尾遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表头向表尾遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREVRANK	从表尾向表头遍历压缩列表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名	从表尾向表头遍历跳跃表，查找给定的成员，沿途记录经过节点的数量，当找到给定成员之后，途经节点的数量就是该成员所对应元素的排名
ZREM	遍历压缩列表，删除所有包含给定成员的节点，以及被删除成员节点旁边的分值节点	遍历跳跃表，删除所有包含了给定成员的跳跃表节点。并在字典中解除被删除元素的成员和分值的关联
ZSCORE	遍历压缩列表，查找包含了给定成员的节点，然后取出成员节点旁边的分值节点保存的元素分值	直接从字典中取出给定成员的分值

8.7 문자열 데이터 타입

Redis 문자열 데이터 타입은 매우 유연합니다.

문자열 데이터 타입은 DEL, EXPIRE, RENAME, TYPE, OBJECT 등의 명령어를 지원합니다.

TYPE 명령어는 문자열 데이터 타입을 반환합니다.

DEL 명령어는 문자열 데이터 타입을 삭제합니다.

```
#
redis> SET msg "hello"
OK
#
redis> RPUSH numbers 1 2 3
(integer) 3
#
redis> SADD fruits apple banana cherry
(integer) 3
redis> DEL msg
(integer) 1
redis> DEL numbers
(integer) 1
redis> DEL fruits
(integer) 1
```

문자열 데이터 타입은 매우 유연합니다.

· SET, GET, APPEND, STRLEN 등의 명령어를 지원합니다.

·HDEL HSET HGET HLEN

·RPUSH LPOP LINSERT LLEN

·SADD SPOP SINTER SCARD

·ZADD ZCARD ZRANK ZSCORE

SET GET APPEND

LLEN Redis

```
redis> SET msg "hello world"
```

```
OK
```

```
redis> GET msg
```

```
"hello world"
```

```
redis> APPEND msg " again!"
```

```
(integer) 18
```

```
redis> GET msg
```

```
"hello world again!"
```

```
redis> LLEN msg
```

```
(error) WRONGTYPE Operation against a key holding the wrong kind of value
```

8.7.1

Redis

redisObject type

· 客户端发送 LLEN 命令

· 服务器检查

key 的值对象是否是列表对象

· 如果是列表对象，服务器返回 LLEN 命令的结果
redisObject 的 type 是 REDIS_LIST 时，返回 LLEN 的结果

· 如果不是列表对象，服务器返回一个类型错误

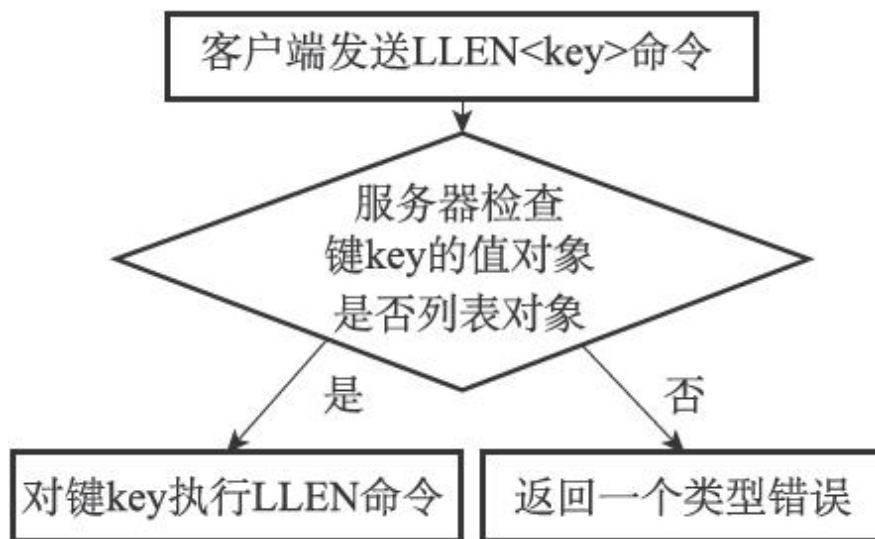


图8-18 LLEN命令执行流程图

客户端发送 LLEN 命令

8.7.2 命令执行

DEL EXPIRE LLEN 命令的语法如下：

命令格式：DEL key

命令格式：EXPIRE key seconds

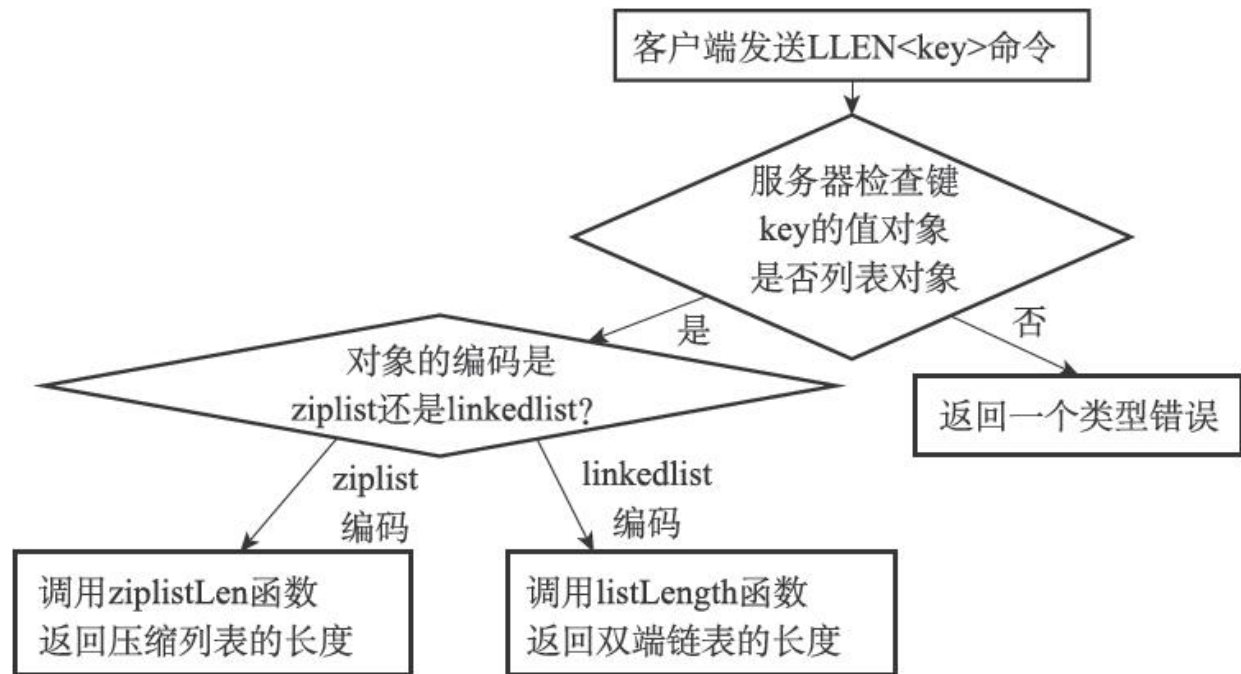


图8-19 LLEN命令逻辑

8.8 参考计数

在C语言中实现Redis的内存管理，
reference counting是其中一种方法。
在Redis中，每个对象都有一个refcount属性。

redisObject的refcount属性

```
typedef struct redisObject {  
    // ...  
    //  
    int refcount;  
    // ...  
} robj;
```

在Redis中，每个对象都有一个refcount属性。

• 当refcount为1时，对象可以被释放。

• 当refcount为0时，对象不能被释放。

• 当refcount为负数时，对象不能被释放。

• 当refcount为0时，对象可以被释放。

8-12 Redis的API和API的调用方法

8-12 字符串API

函数	作用
incrRefCount	将对象的引用计数值增一
decrRefCount	将对象的引用计数值减一，当对象的引用计数值等于 0 时，释放对象
resetRefCount	将对象的引用计数值设置为 0，但并不释放对象，这个函数通常在需要重新设置对象的引用计数值时使用

字符串对象的引用计数值在创建时初始化为 1，在释放时初始化为 0。

字符串对象的引用计数值在释放时初始化为 0。

```
//
// 字符串对象 s
// 字符串对象 1
robj *s = createStringObject(...)
//
// s
// 字符串...
//
// s
// 字符串...0
//
// s
//
decrRefCount(s)
```

字符串对象的引用计数值在释放时初始化为 0。

8.9 面试题

面试官：假设有一个文件，里面存放了100个整数，每个整数都是8-20之间的数。

A：面试官，这个题目我做过，可以用哈希表来解决。

面试官：B：面试官，这个题目我做过，可以用哈希表来解决。

面试官：

1. 面试官：B：面试官，这个题目我做过，可以用哈希表来解决。

2. 面试官：A：面试官，这个题目我做过，可以用哈希表来解决。

面试官：面试官，这个题目我做过，可以用哈希表来解决。

面试官：Redis面试官，这个题目我做过，可以用哈希表来解决。

1. 面试官：面试官，这个题目我做过，可以用哈希表来解决。

2. 面试官：面试官，这个题目我做过，可以用哈希表来解决。

面试官：8-21面试官，这个题目我做过，可以用哈希表来解决。

面试官：面试官，这个题目我做过，可以用哈希表来解决。

面试官：面试官，这个题目我做过，可以用哈希表来解决。

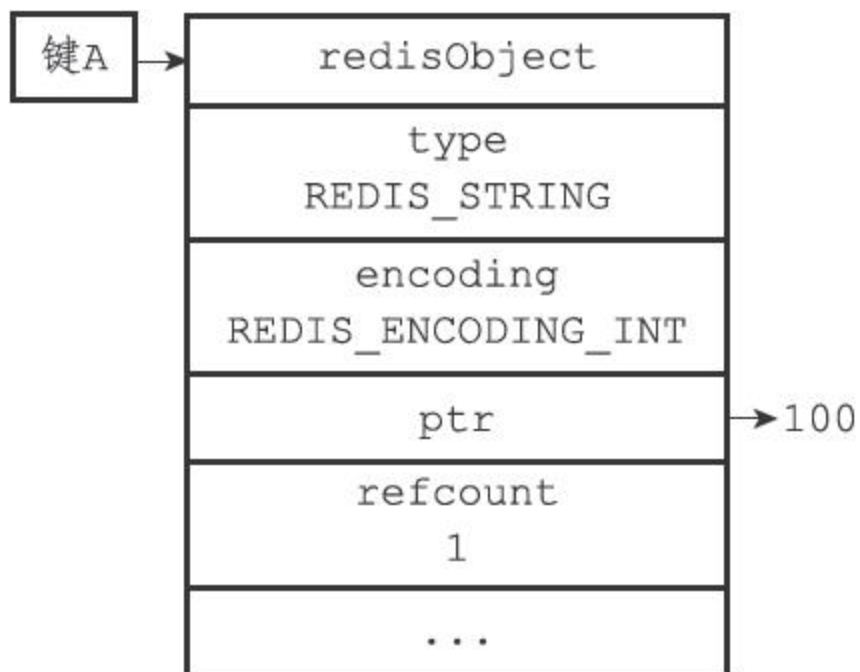


图8-20 内存对象结构

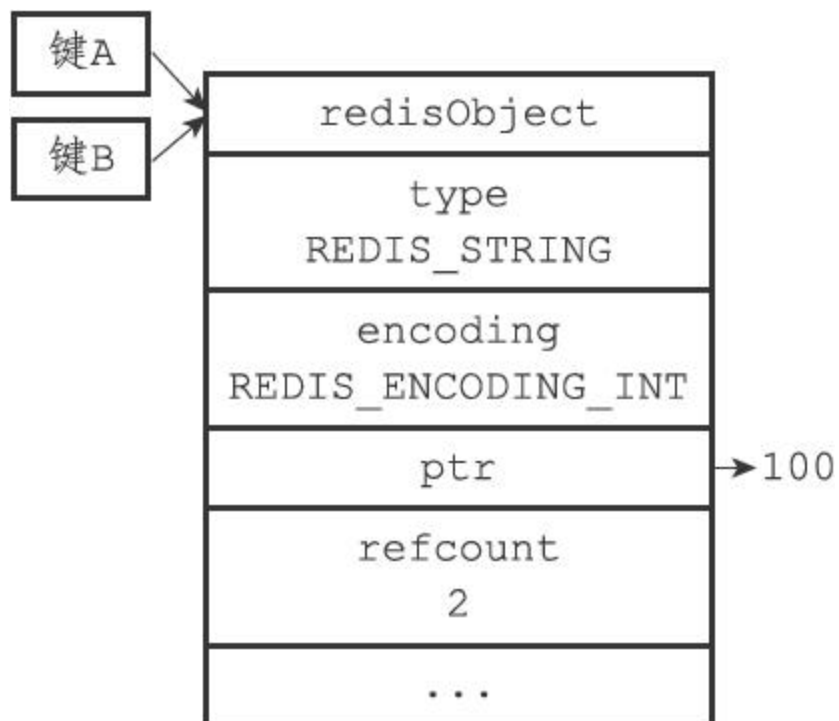


图8-21 内存对象结构

redis-cli 100 A B
redis-cli

redis-cli Redis 0 9999
redis-cli 0 9999
redis-cli



redis.h/REDIS_SHARED_INTEGERS
redis-cli

redis-cli 100 A OBJECT REFCOUNT A
redis-cli 2

```
redis> SET A 100
OK
redis> OBJECT REFCOUNT A
(integer) 2
```

redis-cli A 8-
22

redis-cli 100 B B 100
redis-cli 3

```
redis> SET B 100
OK
redis> OBJECT REFCOUNT A
(integer) 3
redis> OBJECT REFCOUNT B
(integer) 3
```

图8-23 redisObject结构

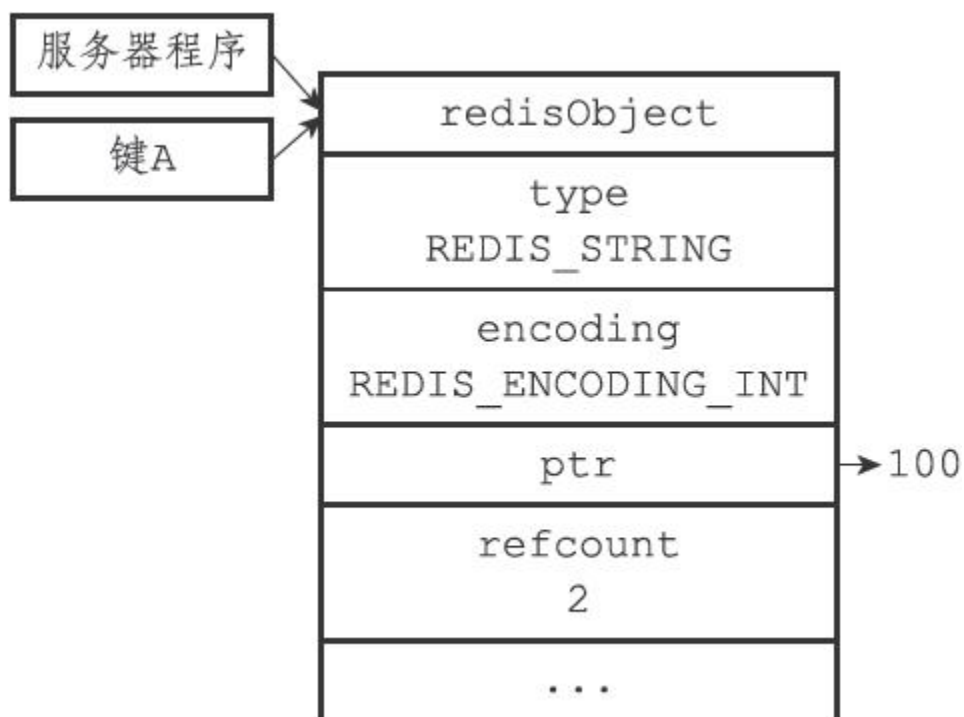


图8-22 redisObject结构

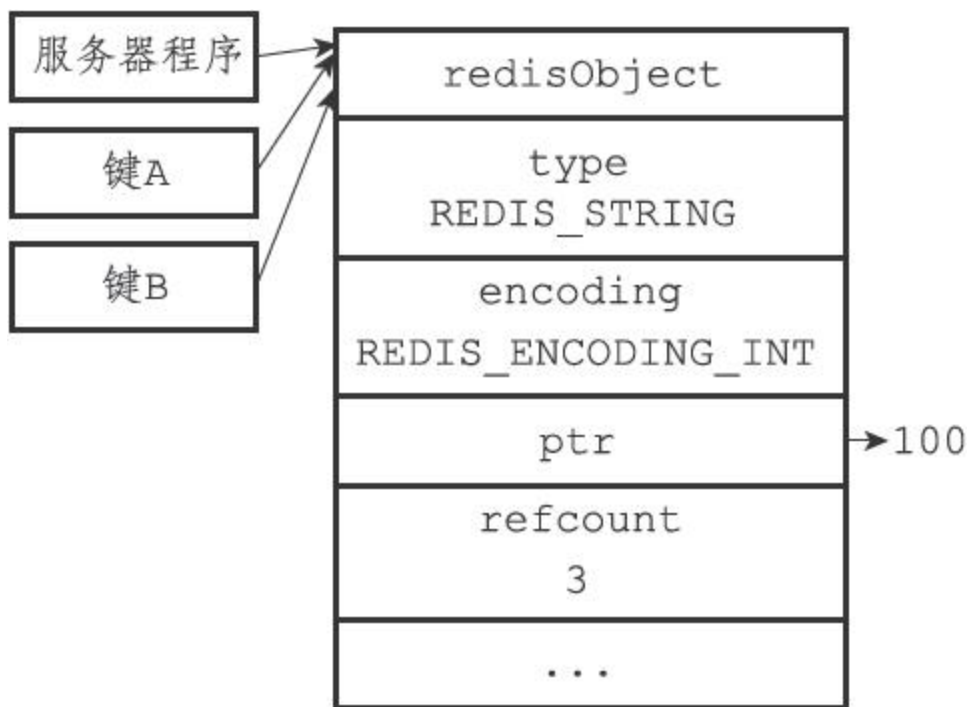


图8-23 redisObject结构

redisObject结构体中，type、encoding、ptr、refcount、...等成员变量，分别表示键的类型、编码、指向、引用计数、...等。其中，type、encoding、ptr、refcount等成员变量，分别对应着redisObject结构体中的type、encoding、ptr、refcount等成员变量。而...则表示键的其他属性，如键的过期时间、键的复制策略等。

redisObject结构体中的type、encoding、ptr、refcount等成员变量，分别对应着redisObject结构体中的type、encoding、ptr、refcount等成员变量。而...则表示键的其他属性，如键的过期时间、键的复制策略等。

redisObject结构体中的type、encoding、ptr、refcount等成员变量，分别对应着redisObject结构体中的type、encoding、ptr、refcount等成员变量。而...则表示键的其他属性，如键的过期时间、键的复制策略等。

Redis的内存管理

Redis的内存管理，是指Redis在运行过程中，对内存的分配、释放、回收等操作。Redis的内存管理，主要分为两部分：内存的分配和内存的回收。Redis的内存分配，主要是通过malloc、calloc、realloc等函数实现的。Redis的内存回收，主要是通过free、freeif等函数实现的。Redis的内存管理，还涉及到内存的碎片化、内存的压缩等问题。

Redis的内存管理，主要分为两部分：内存的分配和内存的回收。Redis的内存分配，主要是通过malloc、calloc、realloc等函数实现的。Redis的内存回收，主要是通过free、freeif等函数实现的。Redis的内存管理，还涉及到内存的碎片化、内存的压缩等问题。

Redis的内存管理，主要分为两部分：内存的分配和内存的回收。Redis的内存分配，主要是通过malloc、calloc、realloc等函数实现的。Redis的内存回收，主要是通过free、freeif等函数实现的。Redis的内存管理，还涉及到内存的碎片化、内存的压缩等问题。

· Redis的内存管理，主要是通过内存池实现的。内存池是一种预先分配好的内存块，Redis在运行过程中，会从内存池中分配内存。内存池的大小，可以根据需要进行调整。Redis的内存管理，还涉及到内存的碎片化、内存的压缩等问题。

- 時間計算量が $O(N)$ である

- 空間計算量が $O(1)$ である

時間計算量が $O(N^2)$ である

時間計算量が $O(N)$ である CPU 時間、Redis 時間

空間計算量が $O(1)$ である

8.10 字典对象

字典对象包含 type、encoding、ptr、refcount 成员。
redisObject 包含 lru 成员。

```
typedef struct redisObject {  
    // ...  
    unsigned lru:22;  
    // ...  
} robj;
```

OBJECT IDLETIME 命令。
字典对象的 lru 成员。

```
redis> SET msg "hello world"  
OK  
#  
字典对象  
redis> OBJECT IDLETIME msg  
(integer) 20  
#  
字典对象  
redis> OBJECT IDLETIME msg  
(integer) 180  
#  
字典对象  
redis> GET msg  
"hello world"  
#  
字典对象  
redis> OBJECT IDLETIME msg  
(integer) 0
```



OBJECT IDLETIME

lru

OBJECT IDLETIME

maxmemory volatile-lru allkeys-

lru maxmemory

maxmemory maxmemory-policy

8.11 練習問題

・Redisのインストールと起動

・Redisのインストールと起動
インストールと起動の方法を確認する。

・Redisのインストールと起動
インストールと起動の方法を確認する。

・Redisのインストールと起動
インストールと起動の方法を確認する。

・Redisのインストールと起動

・Redisのインストールと起動

□□□□ □□□□□□□□

□9□ □□□

□10□ RDB□□□

□11□ AOF□□□

□12□ □□

□13□ □□□

□14□ □□□

9 9

Redis
Redis
2.8

9.1 数据库

Redis 数据库由 redis.h/redisServer 中的 db 结构体
db 结构体由 redis.h/redisDb 中的 redisDb 结构体组成

```
struct redisServer {  
    // ...  
    //  
    redisDb *db;  
    // ...  
};
```

dbnum 是数据库的索引

```
struct redisServer {  
    // ...  
    //  
    int dbnum;  
    // ...  
};
```

dbnum 是数据库的索引，database 是数据库的名称，16 个
Redis 数据库 16 个数据库 9-1

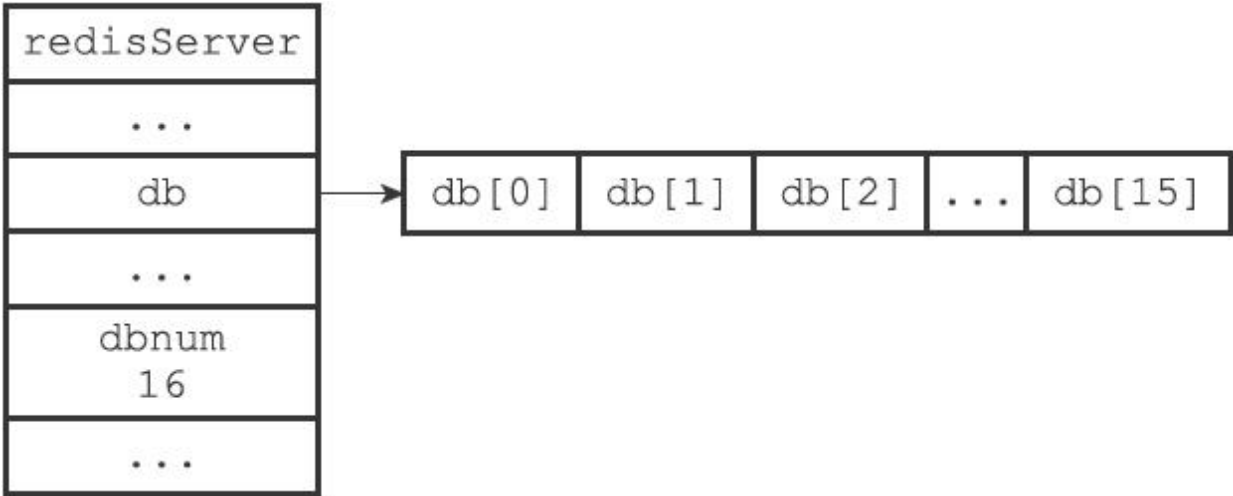


图9-1 Redis服务器结构

9.2 Redis

Redis is a key-value store that can store strings, lists, sets, and more. It is a distributed system that can be used for caching, session storage, and more.

Redis is a key-value store that can store strings, lists, sets, and more. It is a distributed system that can be used for caching, session storage, and more.

Redis is a key-value store that can store strings, lists, sets, and more. It is a distributed system that can be used for caching, session storage, and more.

```
redis> SET msg "hello world"
OK
redis> GET msg
"hello world"
redis> SELECT 2
OK
redis[2]> GET msg
(nil)
redis[2]> SET msg"another world"
OK
redis[2]> GET msg
"another world"
```

Redis is a key-value store that can store strings, lists, sets, and more. It is a distributed system that can be used for caching, session storage, and more.

```
typedef struct redisClient {
// ...
//
```

```

redisDb *db;
// ...
} redisClient;

```

redisClient.db指向redisServer.db指向的数据库

redisServer.db[0]指向的数据库是默认数据库

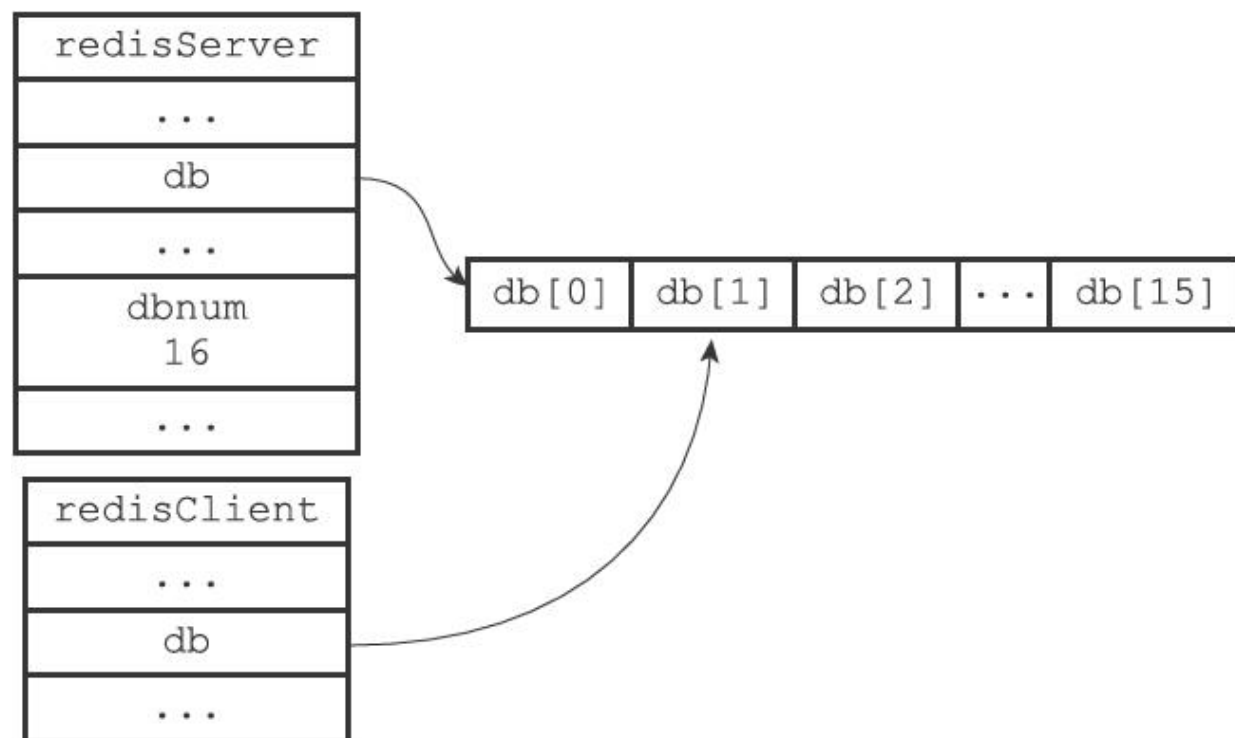


图9-2 数据库的初始化

redisClient.db指向redisServer.db指向的数据库

图9-3

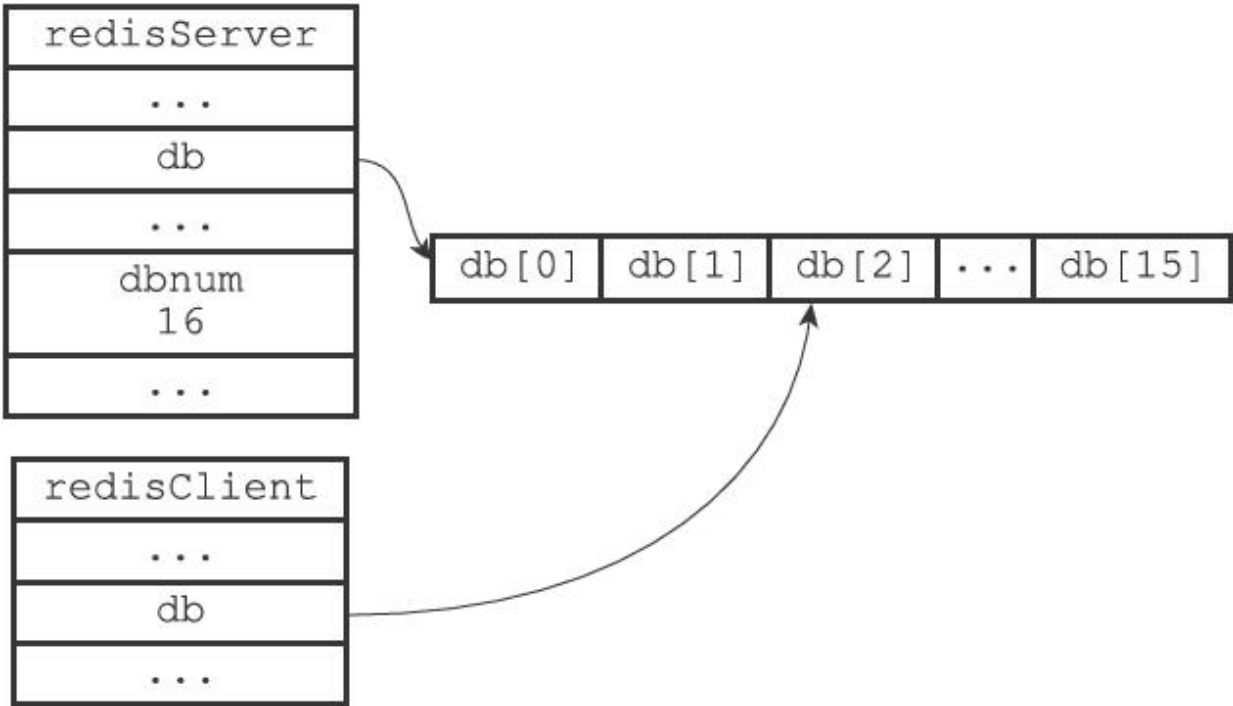


图9-3 Redis数据库2

通过redisClient.db可以访问数据库，通过redisClient.db.select(index)可以切换数据库，通过redisClient.db.select(index)可以切换数据库。

```

Redis客户端redis-cli
redis> SELECT 1
OK
redis[1]> SELECT 2
OK
redis[2]>
  
```

```

redis-cli
redis-cli
Redis
FLUSHDB
SELECT

```

9.3 六六六六六六

```
Redis is a key-value pair database
redis.h/redisDb redisDb dict
key space
```

```
typedef struct redisDb {
    // ...
    //
    dict *dict;
    // ...
} redisDb;
```

[illegible]

■ □

[illegible]

Redis

[illegible]

```
redis> SET message "hello world"
OK
redis> RPush alphabet "a" "b" "c"
(integer)3
redis> HSET book name "Redis in Action"
(integer) 1
redis> HSET book author "Josiah L. Carlson"
```

```
(integer) 1
redis> HSET book publisher "Manning"
(integer) 1
```

Figure 9-4 Redis data structure

·alphabet Redis data structure "alphabet" Redis data structure

·book Redis data structure "book" Redis data structure

·message Redis data structure "message" Redis data structure

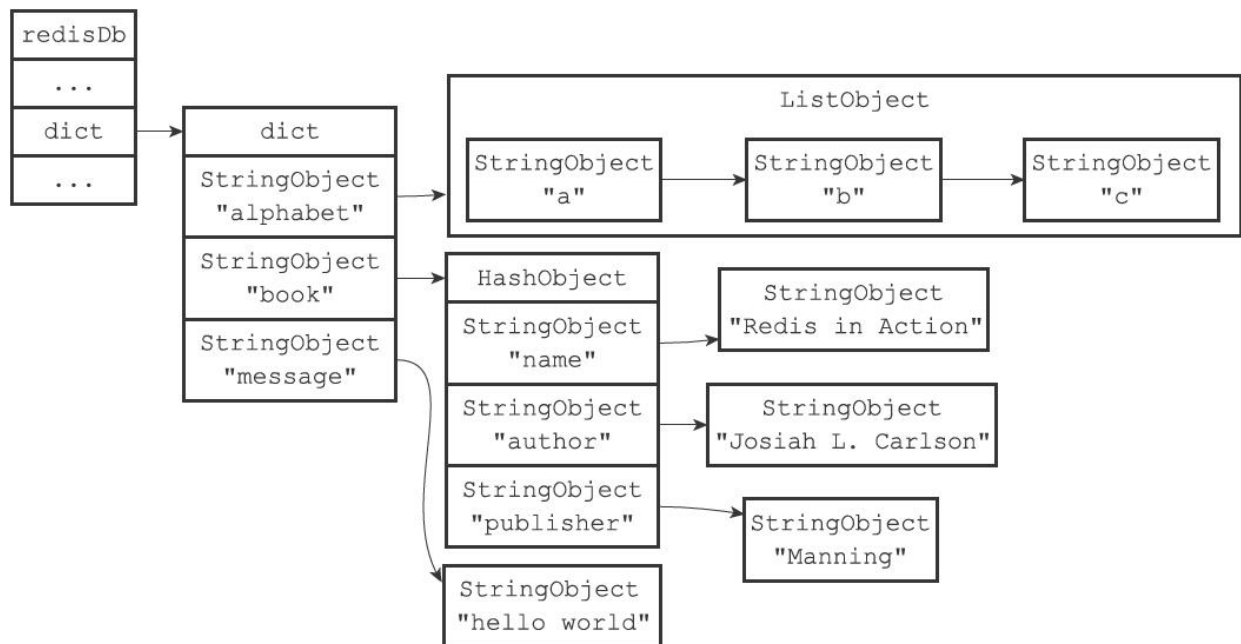


Figure 9-4 Redis data structure

redis> SET date "2013.12.1"
OK
redis> GET date
"2013.12.1"

9.3.1 Redis

redis> SET date "2013.12.1"
OK

redis> GET date

```
redis> SET date "2013.12.1"  
OK
```

redis> GET date
"2013.12.1"

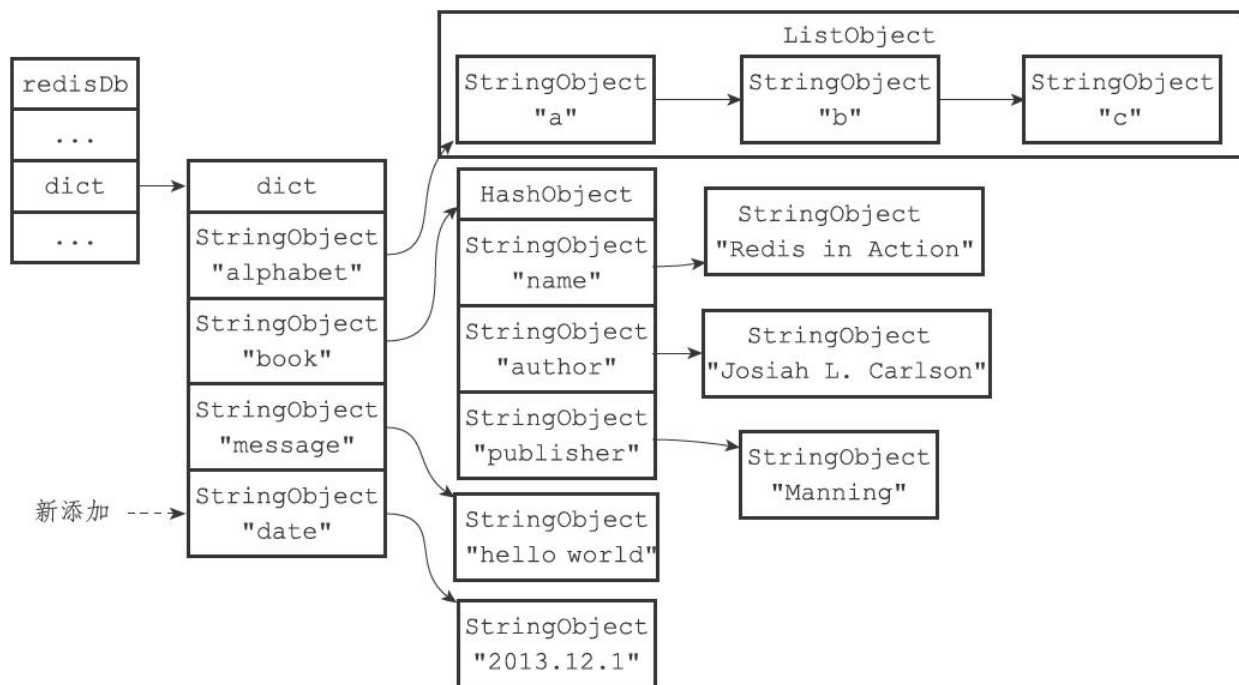


图9-5 Redis中的date数据类型

9.3.2 Redis

Redis是一个开源的分布式键值数据库，它支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、位图等。Redis还支持持久化、主从复制、集群等功能。

Redis的内存模型是基于哈希表的，它使用哈希表来存储键值对。Redis的内存模型支持多种数据类型，包括字符串、列表、集合、有序集合、哈希表、位图等。

```
redis> DEL book
(integer) 1
```

book数据类型为字符串9-6

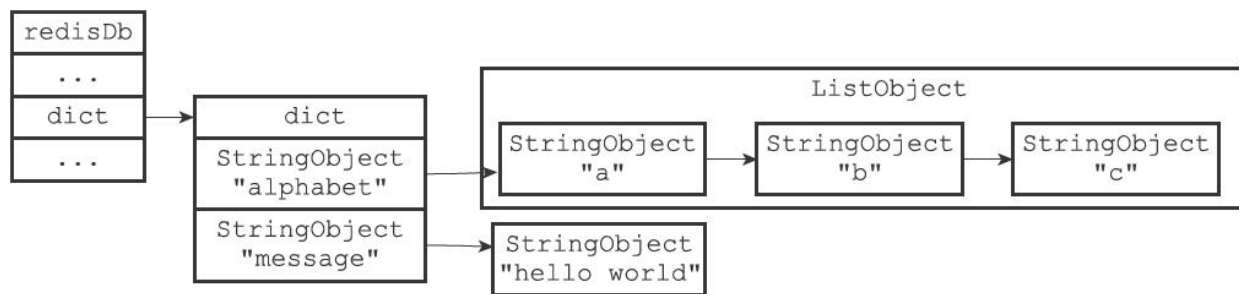


图9-6 字典book的数据结构

9.3.3 列表

列表是 Redis 的另一种数据结构。列表是有序的，并且可以保存任何类型的对象。列表中的对象可以指向其他对象，也可以指向其他列表。列表的底层实现是双向链表。列表的创建和删除操作的时间复杂度是 O(1)。

图9-4展示了列表的数据结构。列表的每个节点都是一个对象，包含一个指向下一个节点的指针和一个指向上一个节点的指针。列表的头指针指向第一个节点，尾指针指向最后一个节点。

```
redis> SET message "blah blah"
OK
```

在 Redis 中，列表的每个节点都是一个对象，包含一个指向下一个节点的指针和一个指向上一个节点的指针。列表的头指针指向第一个节点，尾指针指向最后一个节点。图9-7展示了列表的数据结构。

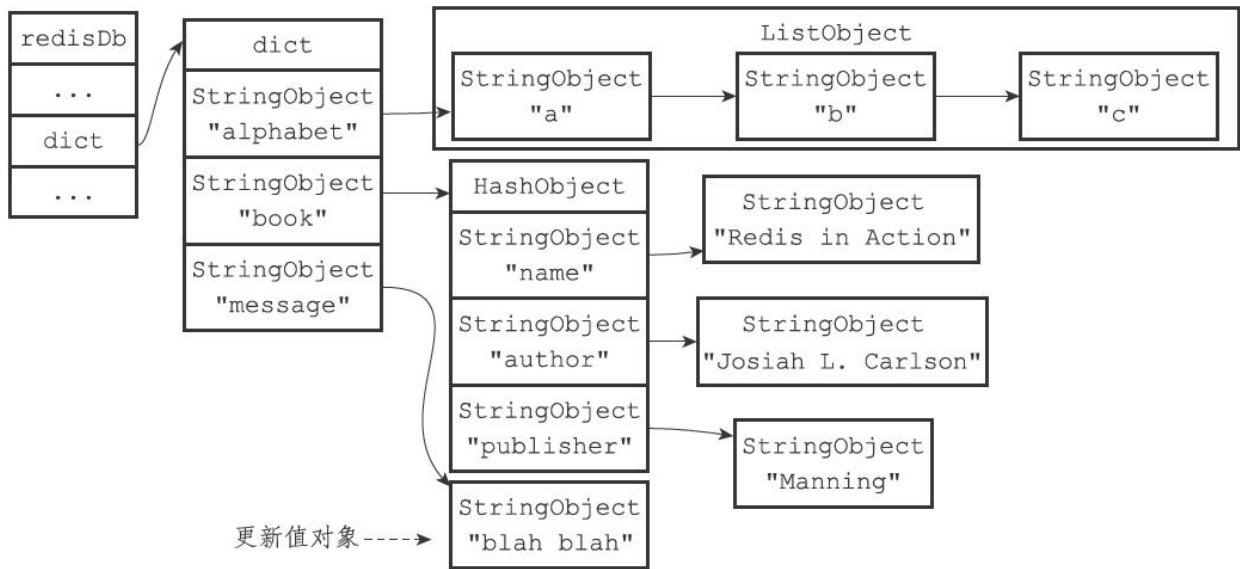


图9-7 使用SET命令设置message

Redis 命令格式

```
redis> HSET book page 320
(integer) 1
```

使用HSET命令将book的page值设置为320

图9-8 Redis命令

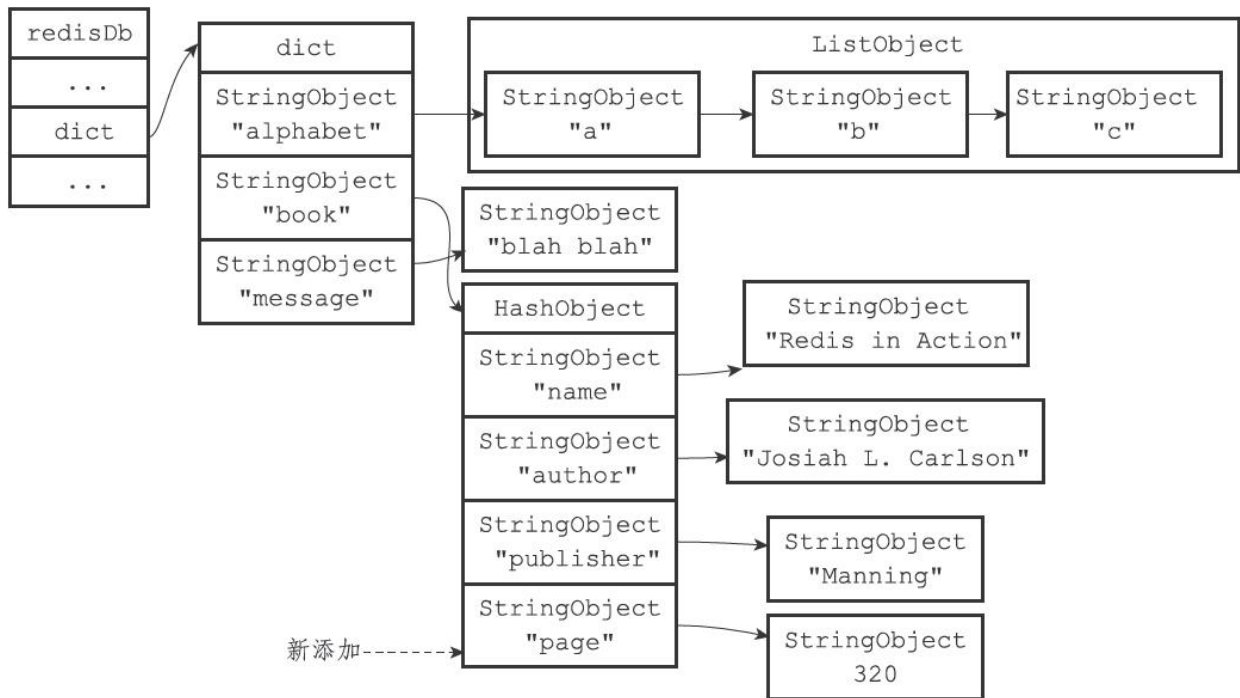


图9-8 HSET命令book

9.3.4 命令

命令的格式如下：

命令的格式如下：9-4

```
redis> GET message
"hello world"
```

GET命令的格式如下：message

命令的格式如下："hello world" 9-9

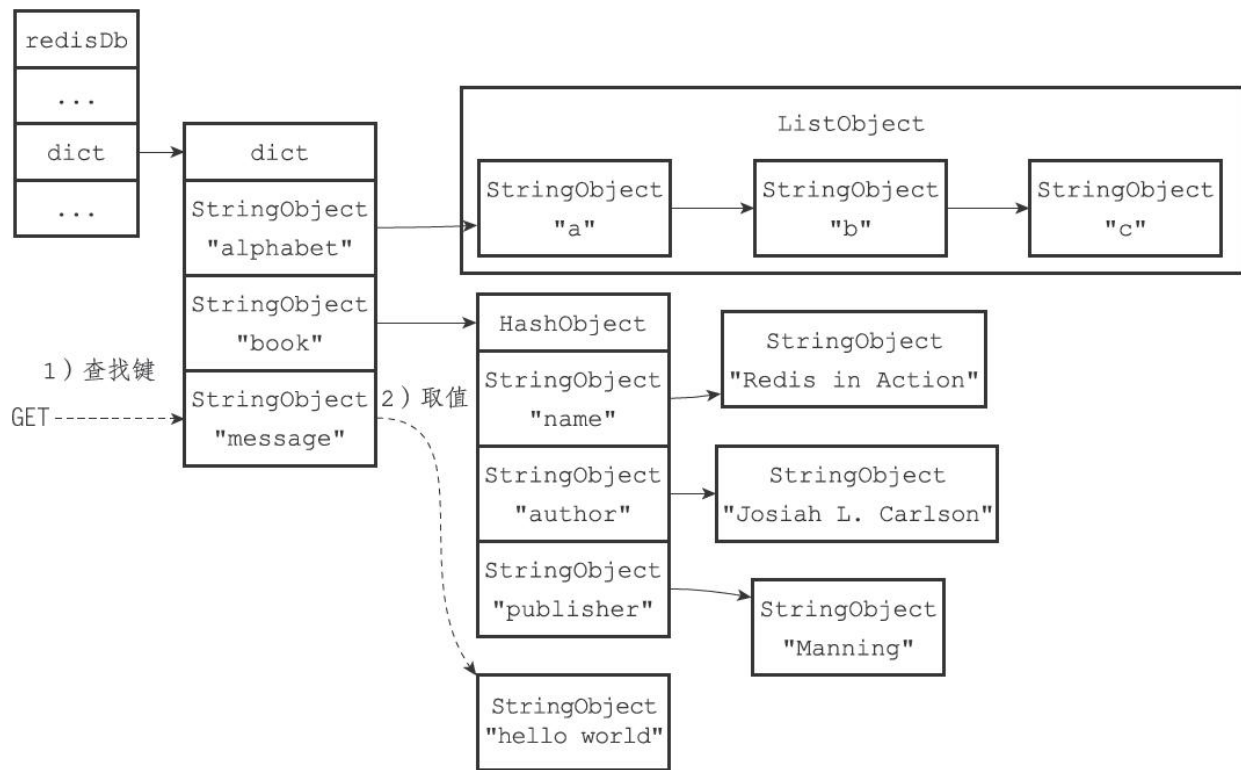


图9-9 GET操作示意图

Redis操作命令

```

redis> LRange alphabet 0 -1
1)"a"
2)"b"
3)"c"
  
```

LRange操作命令返回结果

Redis操作命令9-10

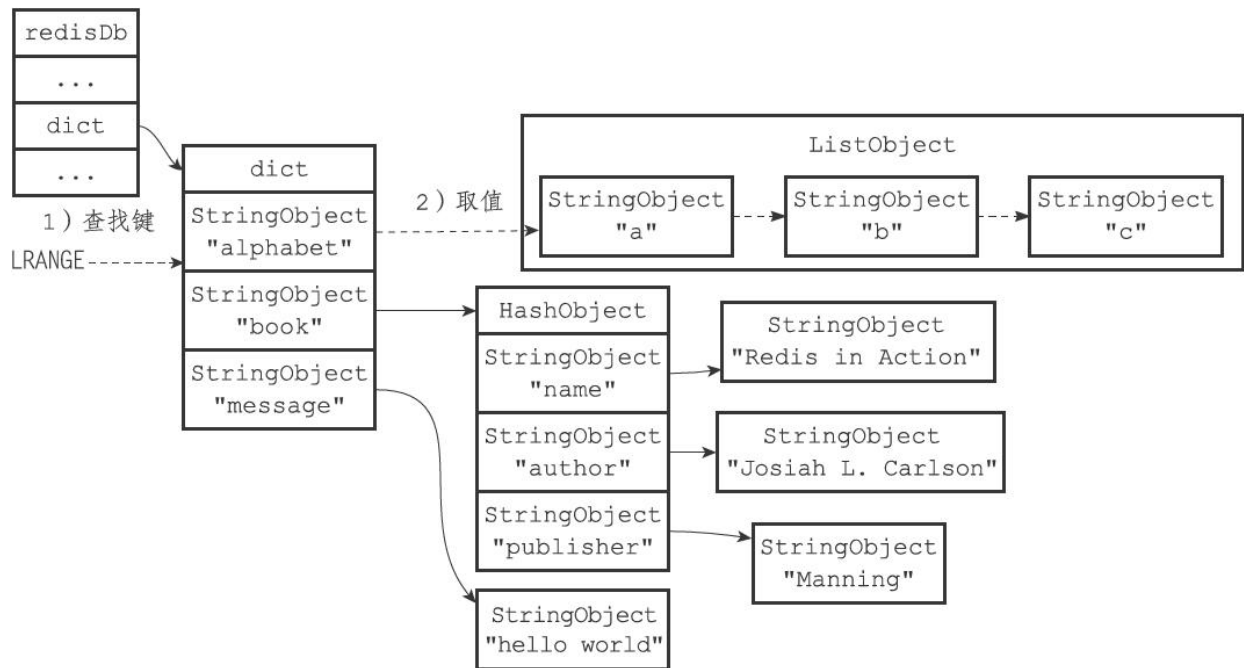


图9-10 LRANGE命令的内部实现

9.3.5 数据库操作

Redis数据库操作命令包括：
`flushdb`：清空数据库

`flushall`：清空所有数据库
`randomkey`：返回数据库中的一个随机键
`dbsize`：返回数据库中的键数

`exists`：判断键是否存在
`rename`：重命名键
`keys`：返回所有匹配的键

9.3.6 字符串操作

Redis 的持久化策略有 RDB 和 AOF 两种，RDB 是快照持久化，AOF 是日志持久化。

Redis 的内存管理策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

Redis 的过期策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

Redis 的过期策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

Redis 的过期策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

Redis 的过期策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

Redis 的过期策略有 LRU 和 LFU 两种，LRU 是最近最少使用，LFU 是最近最少访问。

9.4 过期时间

EXPIRE 和 PEXPIRE 用于为键设置过期时间。过期时间以秒为单位，0 表示永久有效。

```
redis> SET key value
OK
redis> EXPIRE key 5
(integer) 1
redis> GET key // 5
"value"
redis> GET key // 5
(nil)
```



SETEX 用于为键设置过期时间并设置值。SETEX 和 EXPIRE 的用法类似。

EXPIRE 和 PEXPIRE 也可以用于设置过期时间。EXPIREAT 和 PEXPIREAT 用于设置过期时间，expire time 表示过期时间。

UNIX 时间戳用于设置过期时间。

```
redis> SET key value
OK
redis> EXPIREAT key 1377257300
(integer) 1
redis> TIME
1)"1377257296"
2)"296543"
redis> GET key // 1377257300
""
"value"
redis> TIME
1)"1377257303"
2)"230656"
redis> GET key // 1377257300
""
(nil)
```

TTL P TTL

```
redis> SET key value
OK
redis> EXPIRE key 1000
(integer) 1
redis> TTL key
(integer) 997
redis> SET another_key another_value
OK
redis> TIME
1)"1377333070"
2)"761687"
redis> EXPIREAT another_key 1377333100
(integer) 1
redis> TTL another_key
(integer) 10
```

redis-cli -h 127.0.0.1 -p 6379

OK

9.4.1 EXPIRE

Redis EXPIRE 命令用于为指定的 key 设置过期时间。

语法

·EXPIRE <key> <ttl> 为 key 设置过期时间 ttl。

·PEXPIRE <key> <ttl> 为 key 设置过期时间 ttl。

·EXPIREAT <key> <timestamp> 为 key 设置过期时间

timestamp。

·PEXPIREAT <key> <timestamp> 为 key 设置过期时间

timestamp。

Redis 提供了 EXPIRE、PEXPIRE、EXPIREAT、

PEXPIREAT 四个命令，用于为指定的 key 设置过期时间。

其中 EXPIRE 和 PEXPIRE 用于为指定的 key 设置过期时间。

EXPIRE 和 PEXPIRE 的语法如下：

```
def EXPIRE(key,ttl_in_sec):  
    #  
    TTL
```

```
def PEXPIRE(key,ttl_in_sec):
    ttl_in_ms = sec_to_ms(ttl_in_sec)
    PEXPIRE(key,ttl_in_ms)
```

PEXPIREPEXPIREAT

```
def PEXPIRE(key,ttl_in_ms):
    #
    #UNIX
    #
    now_ms = get_current_unix_timestamp_in_ms()
    #
    #TTL
    #
    PEXPIREAT(key,now_ms+ttl_in_ms)
```

EXPIREATPEXPIREAT

```
def EXPIREAT(key,expire_time_in_sec):
    #
    #
    expire_time_in_ms = sec_to_ms(expire_time_in_sec)
    PEXPIREAT(key, expire_time_in_ms)
```

EXPIREPEXPIREEXPIREATPEXPIREAT

9-11

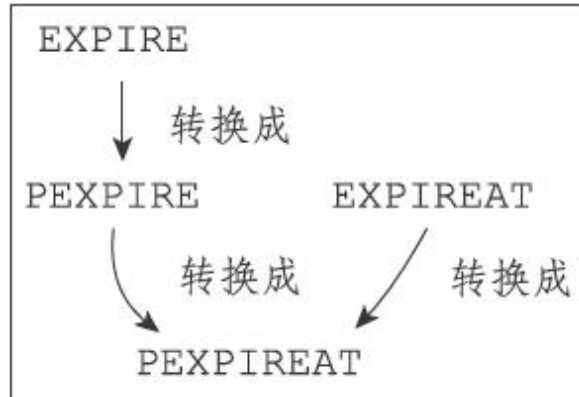


图9-11 命令转换

9.4.2 过期时间

redisDb结构体中expires成员变量用于存储过期时间。

· expires成员变量类型为dict类型，用于存储过期时间。

· expires成员变量类型为long long类型，用于存储过期时间。

在UNIX系统中，

```

typedef struct redisDb {
    // ...
    //
    dict *expires;
    // ...
} redisDb;
  
```

图9-12 redisDb结构体定义

在UNIX系统中，



Figure 9-12 illustrates the internal structure of a Redis database. The database is represented as a table with columns for the key, the value, and the expiration time. The keys are "alphabet" and "book". The values are "1385877600000" and "2013-12-1" respectively. The expiration times are "1385877600000" and "1385560000000" respectively.

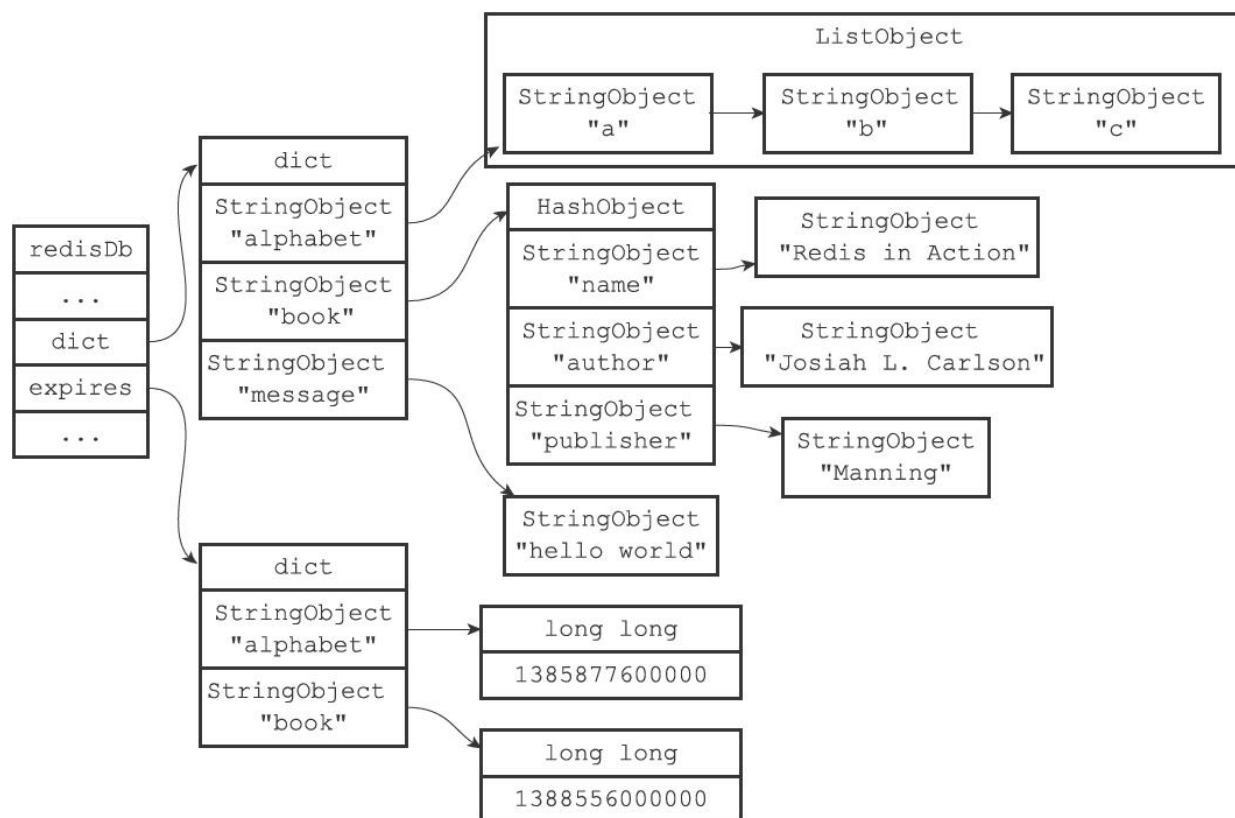


Figure 9-12 Internal structure of a Redis database

Figure 9-12 illustrates the internal structure of a Redis database.

The diagram shows the following structure:

- A table with columns: `redisDb`, `...`, `dict`, `expires`, `...`.
- A `dict` object containing `StringObject "alphabet"`, `StringObject "book"`, and `StringObject "message"`.
- A `dict` object containing `StringObject "alphabet"` and `StringObject "book"`.
- A `StringObject "hello world"` object.
- A `StringObject "a"` object pointing to a `ListObject` containing `StringObject "b"` and `StringObject "c"`.
- A `StringObject "name"` object pointing to a `StringObject "Redis in Action"` object.
- A `StringObject "author"` object pointing to a `StringObject "Josiah L. Carlson"` object.
- A `StringObject "publisher"` object pointing to a `StringObject "Manning"` object.
- A `StringObject "hello world"` object pointing to a `long long` object containing the value `1385877600000`.
- A `StringObject "hello world"` object pointing to a `long long` object containing the value `1385560000000`.

·book138855600000book
138855600000201411

PEXPIREATPEXPIREAT
message139123440000

9-12

```
redis> PEXPIREAT message 139123440000  
(integer) 1
```

message139123440000
2014219-13

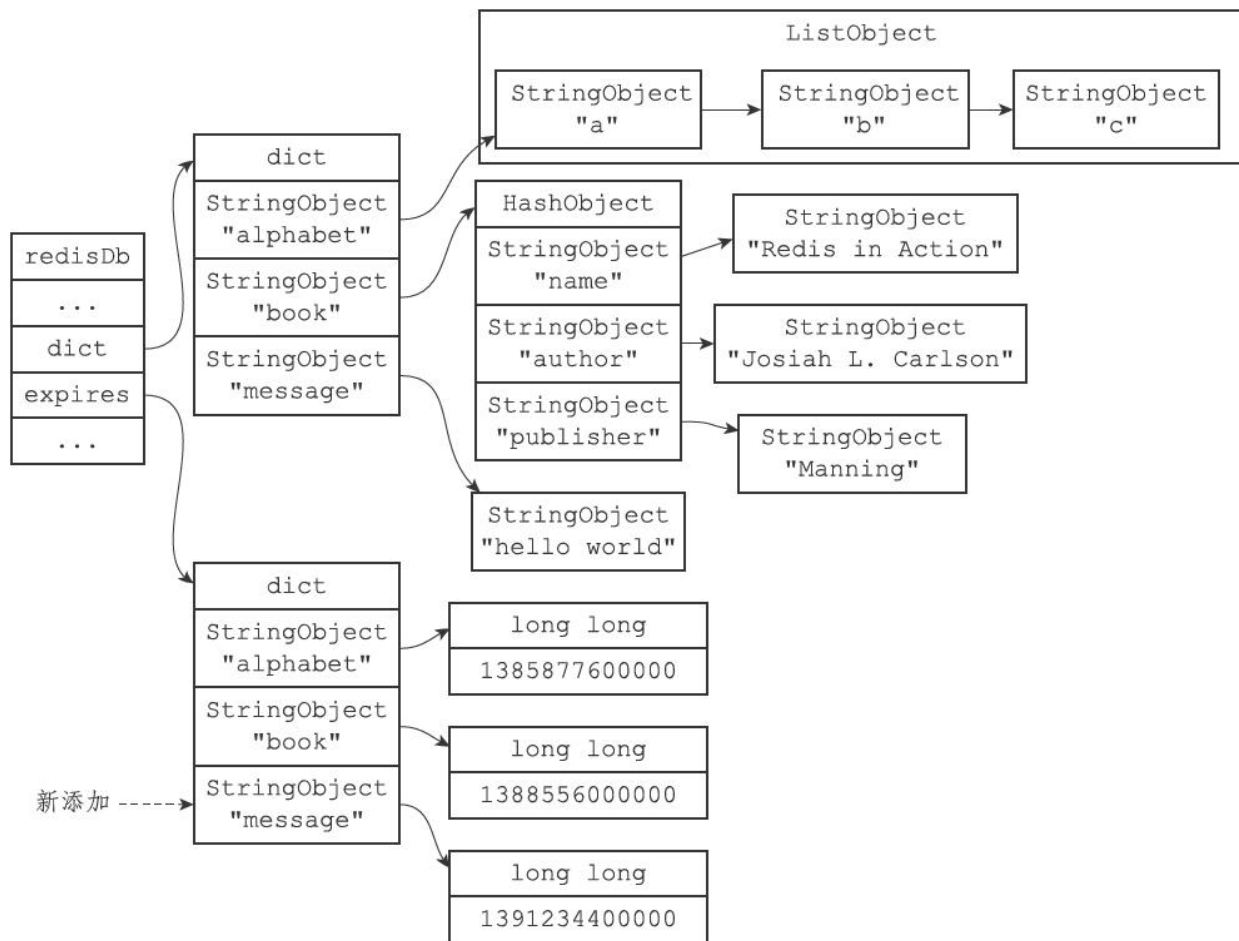


图9-13 PEXPIREAT命令的数据结构

PEXPIREAT命令的实现

```

def PEXPIREAT(key, expire_time_in_ms):
    #
    # 检查key是否存在
    if key not in redisDb.dict:
        return 0
    #
    # 设置过期时间
    redisDb.expires[key] = expire_time_in_ms
    #
    # 返回1
    return 1

```

9.4.3 五五五五五

PERSIST

```
redis> PEXPIREAT message 1391234400000
(integer) 1
redis> TTL message
(integer) 13893281
redis> PERSIST message
(integer) 1
redis> TTL message
(integer) -1
```

PERSIST PEXPIREAT PERSIST

□ □

9-12

```
redis> PERSIST book
(integer) 1
```

□□□□□□□9-14□□□□□□

9.4.4 过期时间

TTL 返回过期时间，PTTL 返回过期时间（毫秒）

```
redis> PEXPIREAT alphabet 1385877600000
(integer) 1
redis> TTL alphabet
(integer) 8549007
redis> PTTL alphabet
(integer) 8549001011
```

TTL 和 PTTL 返回过期时间（毫秒）

过期时间

```
def PTTL(key):
    #
    检查key是否存在
    if key not in redisDb.dict:
        return -2
    #
    检查key是否过期
    #
    返回过期时间（毫秒）
    expire_time_in_ms = None
    expire_time_in_ms = redisDb.expires.get(key)
    #
    检查key是否过期
    if expire_time_in_ms is None:
        return -1
    #
    检查key是否过期
    now_ms = get_current_unix_timestamp_in_ms()
    #
    返回过期时间（毫秒）
    return(expire_time_in_ms - now_ms)
def TTL(key):
    #
    检查key是否存在
    ttl_in_ms = PTTL(key)
```

```

    if ttl_in_ms < 0:
        #
        return -2
    elif ttl_in_ms < -1:
        #
        return -1
    else:
        #
        return ms_to_sec(ttl_in_ms)

```

1385877600000 2013 12 1
 alphabet

· 1383282000000 2013 11 1
 alphabet
 P TTL 2595600000 alphabet
 1385877600000-1383282000000=2595600000

· 1383282000000 2013 11 1
 alphabet TTL 2595600 alphabet
 2595600

9.4.5

1

2

2 UNIX

1

redisDb.expires.get(key)

```
def is_expired(key):
    #
    expire_time_in_ms = redisDb.expires.get(key)
    #
    if expire_time_in_ms is None:
        return False
    #
    now_ms = get_current_unix_timestamp_in_ms()
    #
    if now_ms > expire_time_in_ms:
        #
        return True
    else:
        #
        return False
```

13858776000002013121

alphabet

13832820000002013111

is_expiredalphabetFalsealphabet

13859640000002013122

is_expiredalphabetTruealphabet



```

0000000000000000TTL0000PTTL0000000000000000TTL000

```

0 Redis

```
is_expired[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
```

9.5 数据库

数据库是存储和管理数据的系统。数据库系统由数据库、数据库管理系统、数据库应用程序和数据库用户组成。数据库系统的主要功能是数据定义、数据操作、数据控制和数据通信。

数据库系统的主要功能包括：

- 数据库系统的主要功能包括：数据库定义、数据库操作、数据库控制和数据库通信。

- 数据库系统的主要功能包括：数据库定义、数据库操作、数据库控制和数据库通信。

- 数据库系统的主要功能包括：数据库定义、数据库操作、数据库控制和数据库通信。

数据库系统的主要功能包括：

9.5.1 数据库

数据库是存储和管理数据的系统。数据库系统由数据库、数据库管理系统、数据库应用程序和数据库用户组成。数据库系统的主要功能是数据定义、数据操作、数据控制和数据通信。

1. 在CPU上运行，CPU是系统中最核心的部件，负责执行指令和处理数据。

CPU

Redis — O(N)

[illegible]

9.5.2 〇〇〇〇

CPU

CPU

[illegible][illegible]

9.6 Redis性能优化

Redis性能优化的方法有很多，其中比较重要的有以下几点：
1. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
2. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
3. 使用内存碎片整理（memory defragmentation）来减少内存碎片。

Redis性能优化的方法有很多，其中比较重要的有以下几点：
1. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
2. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
3. 使用内存碎片整理（memory defragmentation）来减少内存碎片。

9.6.1 内存碎片整理

Redis性能优化的方法有很多，其中比较重要的有以下几点：
1. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
2. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
3. 使用内存碎片整理（memory defragmentation）来减少内存碎片。

·使用内存碎片整理（memory defragmentation）来减少内存碎片。

·使用内存碎片整理（memory defragmentation）来减少内存碎片。

Redis性能优化的方法有很多，其中比较重要的有以下几点：
1. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
2. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
3. 使用内存碎片整理（memory defragmentation）来减少内存碎片。

Redis性能优化的方法有很多，其中比较重要的有以下几点：
1. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
2. 使用内存碎片整理（memory defragmentation）来减少内存碎片。
3. 使用内存碎片整理（memory defragmentation）来减少内存碎片。

expireIfNeeded函数
·

-
- expireIfNeeded函数

图9-16 GET函数
·

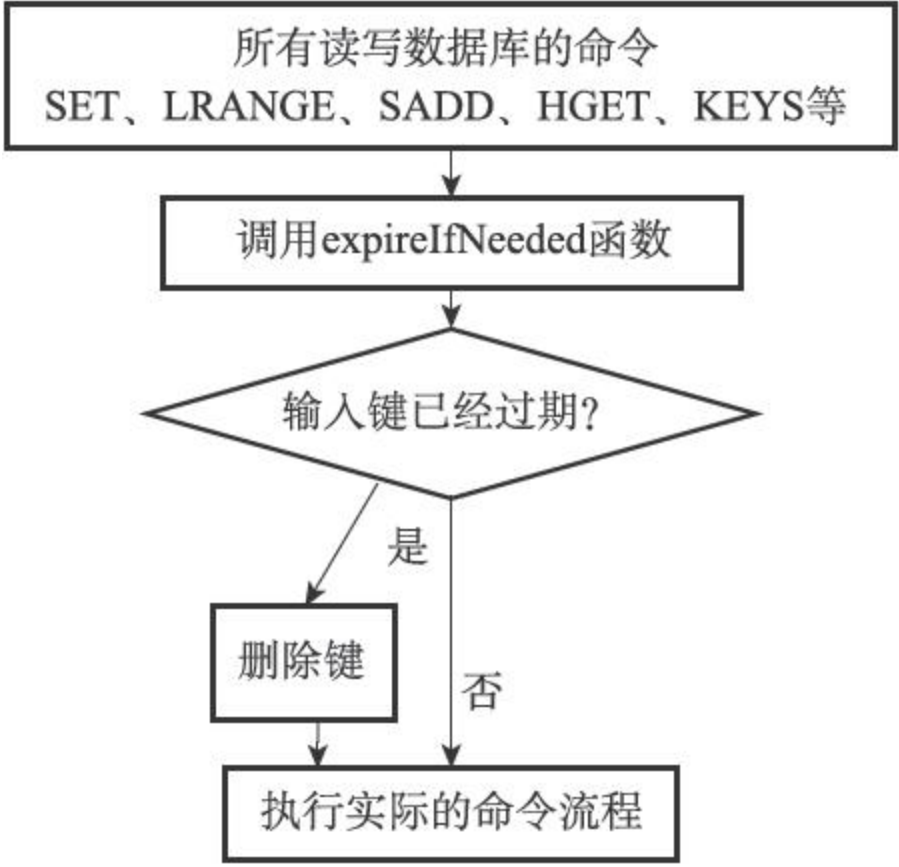


图9-15 调用expireIfNeeded函数

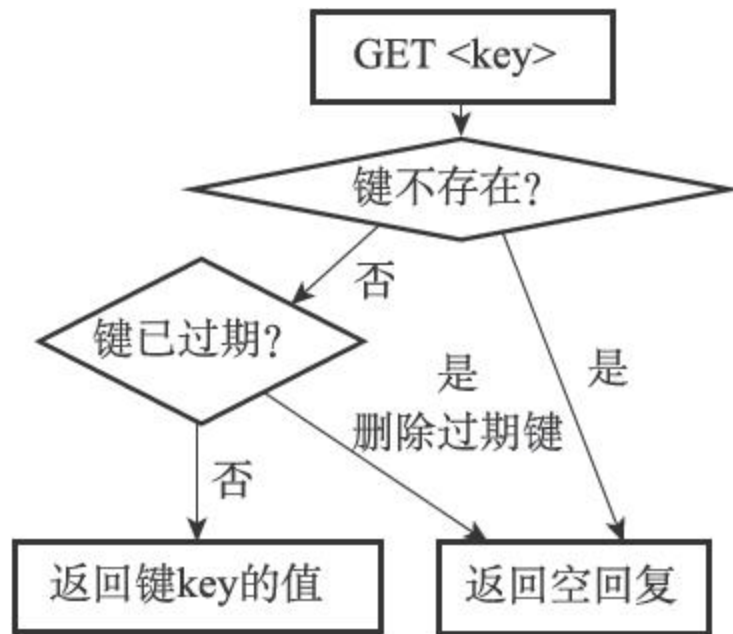


图9-16 GET操作流程图

9.6.2 过期键的删除

Redis的过期键删除是由后台线程定期执行的。在Redis的源代码中，负责删除过期键的函数是`redis.c/activeExpireCycle`。该函数是在`Redis`的主线程中调用的，具体是在`redis.c/serverCron`函数中调用的。该函数会调用`activeExpireCycle`函数，该函数会遍历所有过期键，并调用`expires`函数来删除它们。

下面是一个简单的示例，展示了如何删除过期键。

```

#
# 默认数据库数量
DEFAULT_DB_NUMBERS = 16
#
# 默认键数量
DEFAULT_KEY_NUMBERS = 20
#
# 默认过期时间

```

```

current_db = 0
def activeExpireCycle():
    #
    #####
    #
    ##### DEFAULT_DB_NUMBERS
    ##
    #
    #####
    if server.dbnum < DEFAULT_DB_NUMBERS:
        db_numbers = server.dbnum
    else:
        db_numbers = DEFAULT_DB_NUMBERS
    #
    #####
    for i in range(db_numbers):
        #
        ##current_db
        #####
        #
        #####
        #
        ##current_db
        ##0
        #####
        if current_db == server.dbnum:
            current_db = 0
        #
        #####
        redisDb = server.db[current_db]
        #
        #####1
        #####
        current_db += 1
        #
        #####
        for j in range(DEFAULT_KEY_NUMBERS):
            #
            #####
            if redisDb.expires.size() == 0: break
            #
            #####
            key_with_ttl = redisDb.expires.get_random_key()
            #
            #####
            if is_expired(key_with_ttl):
                delete_key(key_with_ttl)
            #

```

if reach_time_limit(): return

activeExpireCycle

·

·current_dbactiveExpireCycle

activeExpireCycle

activeExpireCycle10activeExpireCycle

11

·activeExpireCycle

current_db0

9.7 AOF与RDB持久化方案对比

Redis持久化方案分为RDB和AOF两种，RDB是快照持久化，AOF是日志持久化。

9.7.1 RDB持久化

Redis的SAVE和BGSAVE命令用于将内存中的数据持久化到磁盘。SAVE命令是同步持久化，BGSAVE命令是异步持久化。

Redis的持久化过程如下：1. 调用SAVE或BGSAVE命令；2. 将内存中的数据写入到磁盘；3. 完成持久化。

Redis的持久化过程如下：1. 调用SAVE或BGSAVE命令；2. 将内存中的数据写入到磁盘；3. 完成持久化。

9.7.2 RDB持久化

Redis的持久化过程如下：1. 调用SAVE或BGSAVE命令；2. 将内存中的数据写入到磁盘；3. 完成持久化。

Redis的持久化过程如下：1. 调用SAVE或BGSAVE命令；2. 将内存中的数据写入到磁盘；3. 完成持久化。

Redis的持久化过程如下：1. 调用SAVE或BGSAVE命令；2. 将内存中的数据写入到磁盘；3. 完成持久化。

Redis RDB

k1 k2 k3 k2

· k1 k3 k2

· k1 k2 k3

9.7.3 AOF

AOF
AOF

AOF append DEL

GET message message

1 message

2 DEL message AOF

3 GET

9.7.4 AOF

RDB AOF
 AOF

$k_1 \leq k_2 \leq k_3$

□□□□□□□□□□□□□□□□AOF□□□□□□□□

9.7.5 □□

[illegible][illegible][illegible]

.XXXXXXXXXXXXDELXXXXXXXXXX

[illegible]

```

message xxx yyy
message 9-17

```



图9-17 初始状态1

客户端向主服务器发送GET message消息，主服务器返回message的值。客户端向从服务器发送message消息，从服务器返回message的值。图9-18

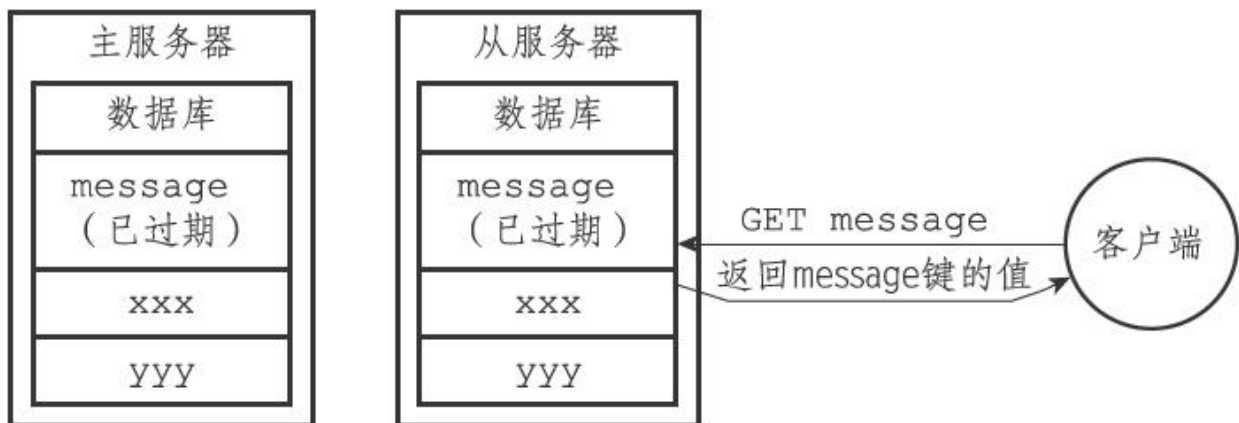


图9-18 初始状态2

客户端向主服务器发送GET message消息，主服务器返回message的值。客户端向从服务器发送message消息，从服务器返回message的值。图9-19

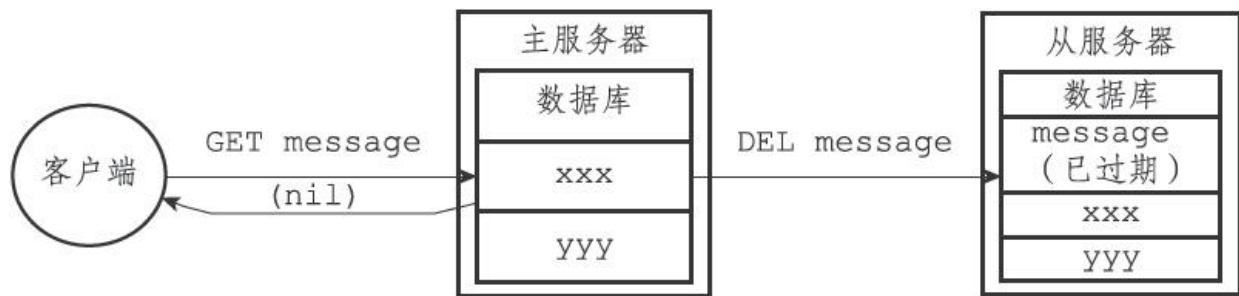


图9-19 图例3

主服务器发送DEL message到从服务器，从服务器删除message。
图9-20

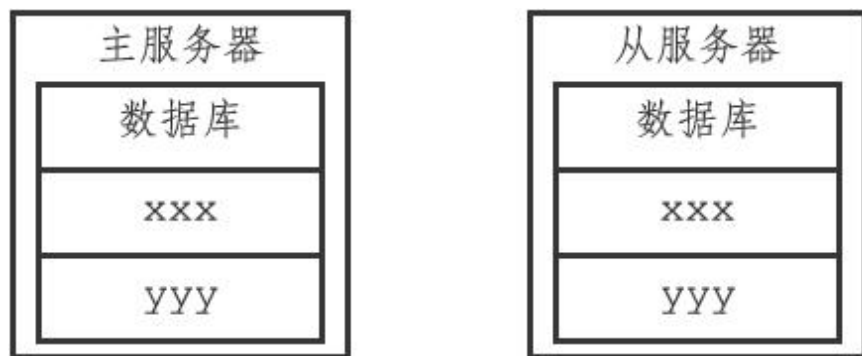


图9-20 图例4

9.8 消息队列

Redis 2.8 版本开始支持消息队列功能，通过 `SUBSCRIBE` 命令可以订阅一个或多个消息队列，通过 `PUBLISH` 命令可以向一个或多个消息队列发布消息。

Redis 消息队列的订阅和发布过程如下：

```
127.0.0.1:6379> SUBSCRIBE __keyspace@0__:message
Reading messages... (press Ctrl-C to quit)
1) "subscribe" //
2) "__keyspace@0__:message"
3) (integer) 1
1) "message" //
2) "set"
3) "set"
1) "message" //
2) "expire"
3) "expire"
1) "message" //
2) "del"
3) "del"
```

Redis 消息队列的订阅和发布过程如下：

1)

redis-cli "redis-cli" --key-space
notification redis-cli key-event notification
redis-cli "redis-cli"

redis-cli 0 DEL

```
127.0.0.1:6379> SUBSCRIBE __keyevent@0__:del
Reading messages... (press Ctrl-C to quit)
1) "subscribe" //
__keyevent@0__:del
2) "__keyevent@0__:del"
3) (integer) 1
1) "message" //
key
DEL
__keyevent@0__:del
2) "__keyevent@0__:del"
3) "key"
1) "message" //
number
DEL
__keyevent@0__:del
2) "__keyevent@0__:del"
3) "number"
1) "message" //
message
DEL
__keyevent@0__:del
2) "__keyevent@0__:del"
3) "message"
```

redis-cli key number message DEL

redis-cli notify-keyspace-events

redis-cli KE

Redis notify-
KeyspaceEvent

SADD saddCommand

```
void saddCommand(redisClient*c){
    // ...
    //
    if (added) {
        // ...
        //
        notifyKeyspaceEvent(REDIS_NOTIFY_SET,"sadd",c->argv[1],c->db->id);
    }
    // ...
}
```

SADD
REDIS_NOTIFY_SET sadd SADD

DEL delCommand

```
void delCommand(redisClient *c){
    int deleted=0,j;
    //
    for (j=1; j<c->argc; j++){
        //
        if (dbDelete(c->db,c->argv[j])){
            // ...
            //
        }
    }
}
```

```

        notifyKeyspaceEvent(REDIS_NOTIFY_GENERIC,
            "del",c->argv[j],c->db->id);
        // ...
    }
}
// ...
}

```

1 delCommand 注册 del 命令
 REDIS_NOTIFY_GENERIC 注册 del 命令
 DEL 注册 del 命令

2 注册 notifyKeyspaceEvent
 saddCommand 注册 delCommand
 notifyKeyspaceEvent 注册 delCommand

9.8.2 注册 del 命令

注册 notifyKeyspaceEvent 命令

```

def notifyKeyspaceEvent(type, event, key, dbid):
    #
    # 注册 del 命令
    if not(server.notify_keyspace_events & type):
        return
    #
    # 注册 del 命令
    if server.notify_keyspace_events & REDIS_NOTIFY_KEYSPACE:
        #
        # 注册 del 命令
        __keyspace@<dbid>__:<key>
        #
        # 注册 del 命令
        <event>
        #
        # 注册 del 命令
        chan = "__keyspace@{dbid}__: {key}".format(dbid=dbid, key=key)

```



```

        #
        pubsubPublishMessage(chan, event)
        #
        if server.notify_keyspace_events & REDIS_NOTIFY_KEYEVENT:
            #
            __keyevent@<dbid>__:<event>
            #
            <key>
            #
            chan = "__keyevent@{dbid}__:
{event}".format(dbid=dbid,event=event)
            #
            pubsubPublishMessage(chan, key)

```

notifyKeyspaceEvent

1 server.notify_keyspace_events notify-
keyspace-events type

2

3

9.9 □□□□

```
·Redis redisServer.db redisServer.dbnum
```

```
· redisServer.db
```

```
·dict expiresdict expires

```

[illegible]

· Redis

[illegible]

```
·Redis[REDACTED]
[REDACTED]
```

```
·  SAVE      BGSAVE      RDB
```

- `BGREWRITEAOF` 命令会重写 `AOF` 文件

- `DEL` 命令删除一个或多个键

- `DEL` 命令删除一个或多个键

- `DEL` 命令删除一个或多个键

Redis 命令大全

- `Redis` 命令大全

10 RDB

Redis是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis还支持持久化，即将内存中的数据保存到磁盘上，以防止数据丢失。

图10-1展示了Redis的持久化机制。Redis服务器包含多个数据库，每个数据库都有自己的持久化文件。当Redis服务器启动时，它会从持久化文件中加载数据到内存中。

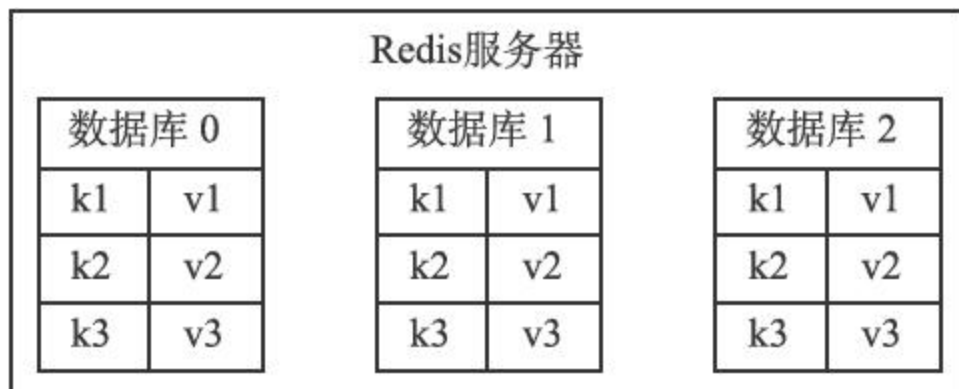


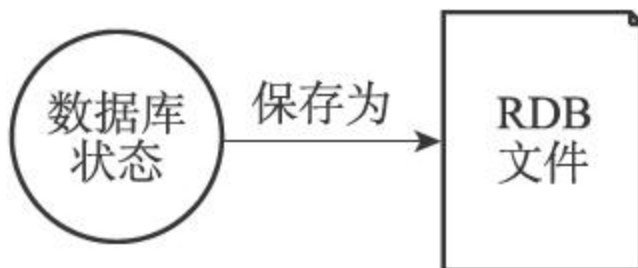
图10-1 Redis持久化机制

Redis的持久化机制主要分为两种：RDB（Redis Database Backup）和AOF（Append Only File）。RDB是将内存中的数据快照保存到磁盘上，而AOF则是将每个写操作记录到日志文件中。

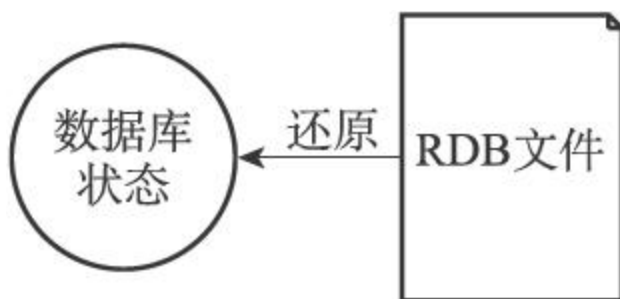
Redis支持多种持久化策略，包括：只使用RDB、只使用AOF、或者同时使用RDB和AOF。用户可以根据需要选择合适的持久化策略。

RDB持久化机制是将内存中的数据快照保存到磁盘上。Redis服务器会定期生成RDB快照，并将它们保存到指定的目录。当Redis服务器启动时，它会从RDB快照中加载数据到内存中。

□□□□□□□□□□10-3□□□



□10-2 □□□□□□□□RDB□□



□10-3 □RDB□□□□□□□□

```

RDBRedisRedis
RDBRedis

```

```

redis-cli -h 127.0.0.1 -p 6379
redis> flushdb
redis> flushall
redis> save
redis> bgsave
redis>

```

Redis

RDB

□□□□□□□□□□RDB□□□□□□□□□□□□□□RDB□□□□□□□□

□□□□□□

10.1 RDB快照持久化

Redis提供了两种持久化方式：RDB和AOF。其中，RDB（Redis Database Backup）是快照持久化，而AOF（Append Only File）是日志持久化。

SAVE命令用于同步地将当前内存中的数据集写入磁盘，生成一个RDB快照文件。

该命令的语法如下：

```
redis> SAVE //
同步RDB
快照成功
OK
```

BGSAVE命令用于异步地将当前内存中的数据集写入磁盘，生成一个RDB快照文件。

该命令的语法如下：

```
redis> BGSAVE //
异步RDB
快照成功
Background saving started
```

在Redis 3.0之前，RDB快照是由主进程（redis-server）直接调用的。从Redis 3.0开始，RDB快照是由子进程（redis-bgsave）调用的。

在Redis 3.0之前，RDB快照是由主进程（redis-server）直接调用的。从Redis 3.0开始，RDB快照是由子进程（redis-bgsave）调用的。

```
def SAVE():
    #
    同步RDB
    快照成功
    rdbSave()
def BGSAVE():
```

```

#
main()
    pid = fork()
    if pid == 0:
        #
        saveRDB
        rdbSave()
        #
        signal_parent()
    elif pid > 0:
        #
        handle_request_and_wait_signal()
    else:
        #
        handle_fork_error()

```

SAVE BGSAVE RDB RDB
 Redis RDB Redis
 RDB RDB

Redis DB loaded from disk: ...
 RDB

```

$ redis-server
[7379] 30 Aug 21:07:01.270 # Server started, Redis version 2.9.11
[7379] 30 Aug 21:07:01.289 * DB loaded from disk: 0.018 seconds
[7379] 30 Aug 21:07:01.289 * The server is now ready to accept connections
on port 6379

```

AOF RDB

· 每次写入数据时，先将数据写入 AOF 文件，再将数据写入 RDB 文件

· 每次写入数据时，先将数据写入 RDB 文件，再将数据写入 AOF 文件

图 10-4 持久化功能流程图

图 10-5 RDB 文件与数据库状态的交互

图 10-5 RDB 文件与数据库状态的交互

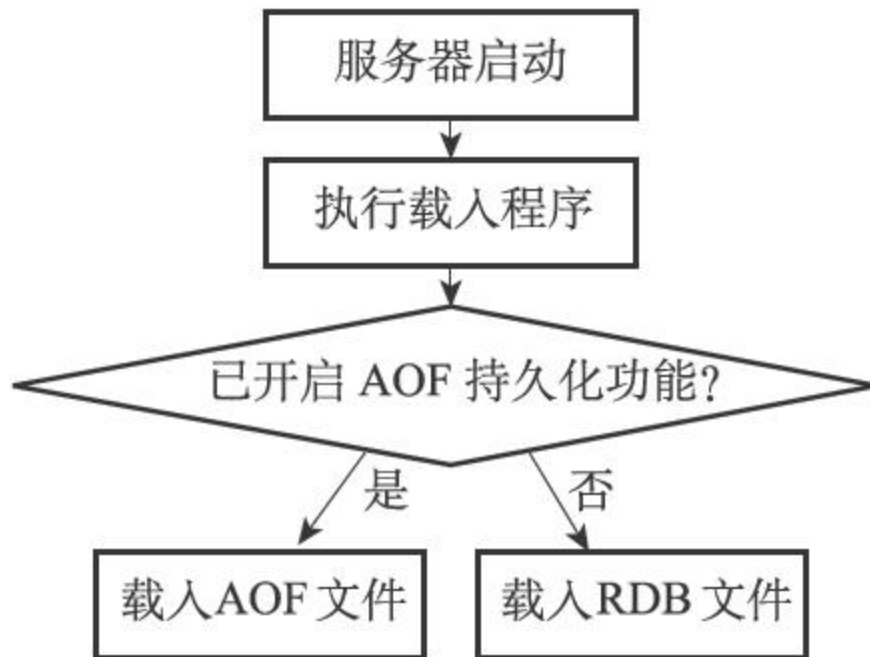


图 10-4 持久化功能流程图

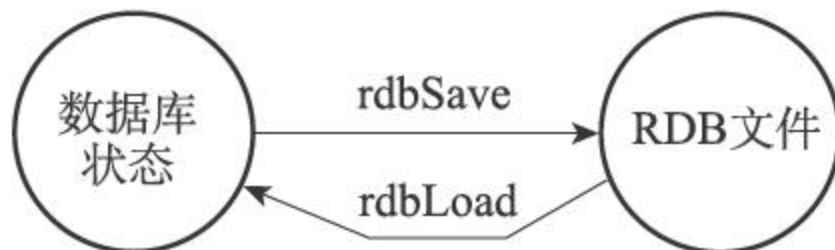


图 10-5 RDB 文件与数据库状态的交互

10.1.1 SAVE 命令

Redis 的 `SAVE` 命令会阻塞 Redis 服务，直到数据被安全地保存到磁盘。这个命令通常用于在 Redis 实例关闭前，将数据保存到磁盘。

Redis 的 `SAVE` 命令会阻塞 Redis 服务，直到数据被安全地保存到磁盘。

10.1.2 BGSAVE 命令

Redis 的 `BGSAVE` 命令会在后台线程中保存 Redis 数据到磁盘。这个命令不会阻塞 Redis 服务，因此可以在 Redis 实例运行时使用。Redis 的 `BGSAVE` 命令会启动一个后台线程，该线程会复制 Redis 的内存数据到磁盘上的 RDB 文件中。

Redis 的 `BGSAVE` 命令会在后台线程中保存 Redis 数据到磁盘。这个命令不会阻塞 Redis 服务，因此可以在 Redis 实例运行时使用。Redis 的 `BGSAVE` 命令会启动一个后台线程，该线程会复制 Redis 的内存数据到磁盘上的 RDB 文件中。

Redis 的 `BGSAVE` 命令会在后台线程中保存 Redis 数据到磁盘。这个命令不会阻塞 Redis 服务，因此可以在 Redis 实例运行时使用。Redis 的 `BGSAVE` 命令会启动一个后台线程，该线程会复制 Redis 的内存数据到磁盘上的 RDB 文件中。

Redis 的 `BGREWRITEAOF` 命令会在后台线程中重写 Redis 的 AOF 文件。这个命令不会阻塞 Redis 服务，因此可以在 Redis 实例运行时使用。

Redis 的 `BGREWRITEAOF` 命令会在后台线程中重写 Redis 的 AOF 文件。这个命令不会阻塞 Redis 服务，因此可以在 Redis 实例运行时使用。

·BGREWRITEAOFBGSAVE

BGREWRITEAOFBGSAVE
——

10.1.3 RDB

RDB

10.2 Redisのバックアップ

Redisは、`SAVE`と`BGSAVE`の2つのコマンドでバックアップを実行する。
`SAVE`は、バックグラウンドでバックアップを実行する。`BGSAVE`は、バックグラウンドでバックアップを実行する。`SAVE`は、バックグラウンドでバックアップを実行する。`BGSAVE`は、バックグラウンドでバックアップを実行する。

`BGSAVE`は、バックグラウンドでバックアップを実行する。Redisは、バックグラウンドでバックアップを実行する。`save`は、バックグラウンドでバックアップを実行する。`BGSAVE`は、バックグラウンドでバックアップを実行する。

`save`は、バックグラウンドでバックアップを実行する。`BGSAVE`は、バックグラウンドでバックアップを実行する。

バックアップの実行方法

```
save 900 1
save 300 10
save 60 10000
```

バックアップの実行方法

- ・`save 900 1`は、バックグラウンドでバックアップを実行する。
- ・`save 300 10`は、バックグラウンドでバックアップを実行する。
- ・`save 60 10000`は、バックグラウンドでバックアップを実行する。

Redis每隔60秒自动进行一次10000个数据的BGSAVE操作

```
[5085] 03 Sep 17:09:49.463 * 10000 changes in 60 seconds. Saving...
[5085] 03 Sep 17:09:49.463 * Background saving started by pid 5189
[5189] 03 Sep 17:09:49.522 * DB saved on disk
[5189] 03 Sep 17:09:49.522 * RDB: 0 MB of memory used by copy-on-write
[5085] 03 Sep 17:09:49.563 * Background saving terminated with success
```

Redis的save操作

10.2.1 配置

Redis的save操作

```
save 900 1
save 300 10
save 60 10000
```

Redis的save操作

```
struct redisServer {
    // ...
    //
    struct saveparam *saveparams;
};
```

```
// ...  
};
```

```
saveparams[]saveparam[]  
saveparam[]save[]
```

```
struct saveparam {  
    //  
    time_t seconds;  
    //  
    int changes;  
};
```

```
save[]
```

```
save 900 1  
save 300 10  
save 60 10000
```

```
saveparams[]10-6[]
```

redisServer		saveparams[0]	saveparams[1]	saveparams[2]
...		seconds	seconds	seconds
saveparams	→	900	300	60
...		changes	changes	changes
		1	10	10000

10-6 []

10.2.2 dirty和lastsave

在saveparams中增加dirty和lastsave两个成员

·dirty表示是否已经脏，SAVE和BGSAVE都会设置dirty为1，表示已经脏

·lastsave表示最后一次成功保存的时间，SAVE和BGSAVE都会设置lastsave为当前时间

```
struct redisServer {  
    // ...  
    //  
    long long dirty;  
    //  
    time_t lastsave;  
    // ...  
};
```

在redisServer中增加dirty和lastsave两个成员

在redisServer中增加dirty和lastsave两个成员

```
redis> SET message "hello"  
OK
```

redis> SADD dirty 1

redis> TYPE dirty

redis> SADD database Redis MongoDB MariaDB
(integer) 3

redis> SADD dirty 3

redisServer
...
dirty 123
lastsave 1378270800
...

图10-7 Redis数据库结构

图10-7展示了Redis数据库的结构，其中dirty和lastsave是两个重要的属性。

·dirty属性表示数据库是否脏，即是否有未同步的数据。在图中，dirty的值为123。

·lastsave属性表示最后一次保存的时间戳。在图中，lastsave的值为1378270800，即2013年9月4日。

图10-7

10.2.3 Redis数据库结构

Redis每隔100ms调用serverCron函数，每隔100ms调用一次save函数，每隔100ms调用一次BGSAVE函数。

serverCron函数实现如下：

```
def serverCron():
    # ...
    #
    for saveparam in server.saveparams:
        #
        save_interval = unixtime_now()-server.lastsave
        #
        #
        #
        if server.dirty >= saveparam.changes and \
            save_interval > saveparam.seconds:
            BGSAVE()
    # ...
```

saveparams函数实现如下：

Redis每隔10-8ms调用一次

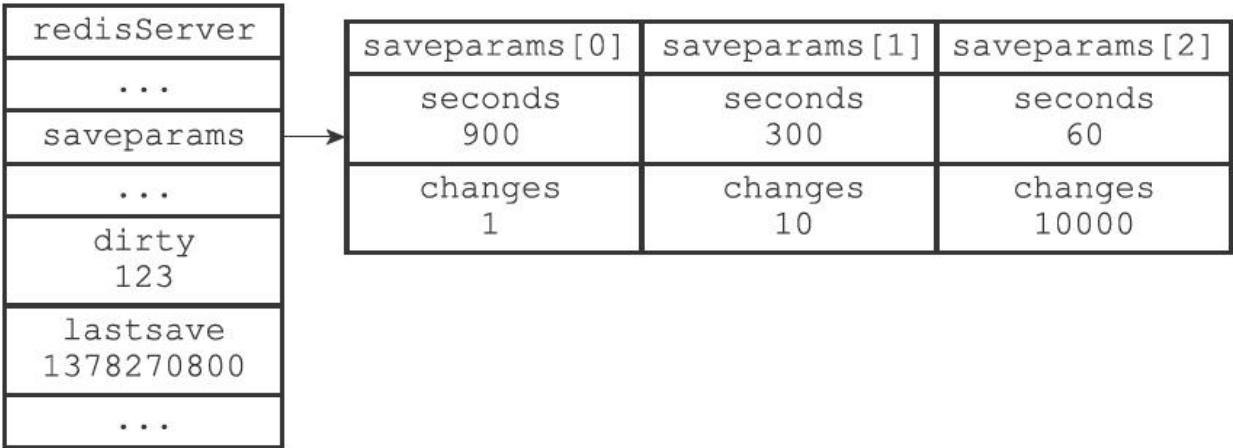


图10-8 初始配置

redisServer的lastsave为1378271101，saveparams[0]为900，saveparams[1]为300，saveparams[2]为60。当执行BGSAVE时，redisServer的dirty为123，lastsave为1378270800。此时，redisServer的saveparams[0]为900，saveparams[1]为300，saveparams[2]为60。

当执行BGSAVE时，redisServer的dirty为0，lastsave为1378271106。此时，redisServer的saveparams[0]为900，saveparams[1]为300，saveparams[2]为60。

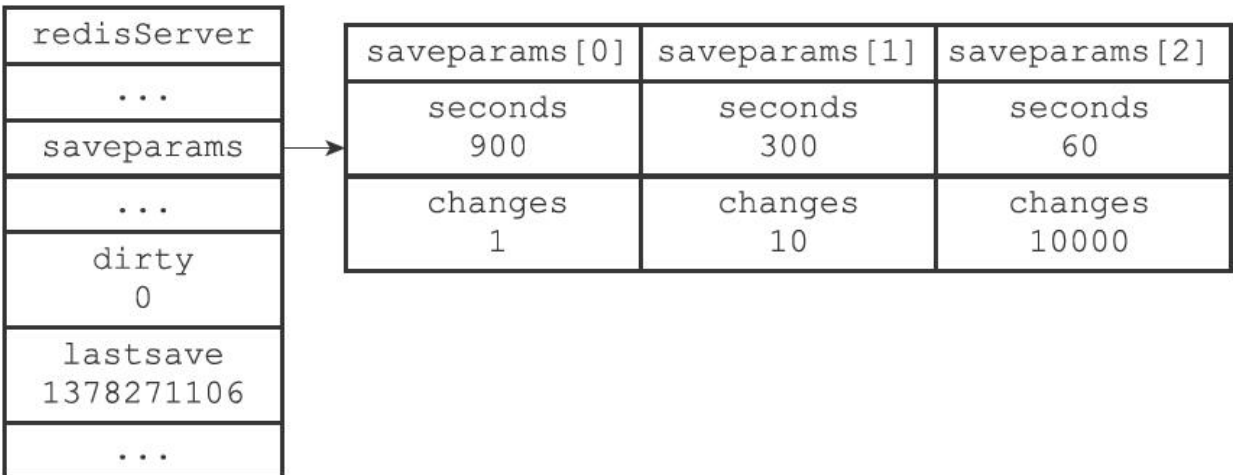


图10-9 BGSAVE后的配置

Redis save BGSAVE
Redis save BGSAVE

10.3 RDB

Redis RDB

10-10 RDB

REDIS	db_version	databases	EOF	check_sum
-------	------------	-----------	-----	-----------

10-10 RDB



10-10 RDB

RDB REDIS 5 "REDIS" RDB



RDB C "REDIS" 'R' 'E' 'D' 'I' 'S' '\0' C

'R'E'D'I'S'\0'RDB

db_version4RDB"0006"RDBRDB

databases
.
.
0
RDB

EOF1RDB

check_sum8
REDISdb_versiondatabasesEOF
RDBcheck_sum
RDB

10-11databasesRDB
"REDIS"RDB"0006"RDB
databasesEOF
6265312314761917404

"REDIS"	"0006"	EOF	6265312314761917404
---------	--------	-----	---------------------

图10-11 databases RDB

10.3.1 databases

RDB databases

0310-12 RDB
database 0000 database 3333
3333

REDIS	db_version	database 0	database 3	EOF	check_sum
-------	------------	------------	------------	-----	-----------

图10-12 RDB

RDB SELECTDB db_number
key_value_pairs10-13

SELECTDB	db_number	key_value_pairs
----------	-----------	-----------------

图10-13 RDB

SELECTDB1
3333

db_number 1 2 3 4 5
 db_number SELECT

key_value_pairs
 key_value_pairs

10-14 RDB 0

SELECTDB	0	key_value_pairs
----------	---	-----------------

10-14

10-15 RDB 0 3

REDIS	db_version	SELECTDB	0	pairs	SELECTDB	3	pairs	EOF	check_sum
-------	------------	----------	---	-------	----------	---	-------	-----	-----------

10-15 RDB

10.3.2 key_value_pairs

RDB key_value_pairs

RDB TYPE key value 10-16

TYPE	key	value
------	-----	-------

10-16 1000000000

TYPE value 1

·REDIS_RDB_TYPE_STRING

·REDIS_RDB_TYPE_LIST

·REDIS_RDB_TYPE_SET

·REDIS_RDB_TYPE_ZSET

·REDIS_RDB_TYPE_HASH

·REDIS_RDB_TYPE_LIST_ZIPLIST

·REDIS_RDB_TYPE_SET_INTSET

·REDIS_RDB_TYPE_ZSET_ZIPLIST

·REDIS_RDB_TYPE_HASH_ZIPLIST

TYPE RDB

TYPE value key value

·key Redis RDB_TYPE_STRING
value key

·TYPE value
TYPE value

RDB 10-17

EXPIRETIME_MS	ms	TYPE	key	value
---------------	----	------	-----	-------

10-17

TYPE key value
TYPE key value EXPIRETIME_MS ms

·EXPIRETIME_MS 1

·ms 8 UNIX

REDIS_RDB_TYPE_STRING	key	value
-----------------------	-----	-------

10-18

10-18

10-19 138855600000

2014 1 1

EXPIRETIME_MS	138855600000	REDIS_RDB_TYPE_SET	key	value
---------------	--------------	--------------------	-----	-------

10-19

10.3.3 value

RDB value TYPE

value

RDB



REDIS_ENCODING_* 8

1.

TYPE REDIS_RDB_TYPE_STRING value

REDIS_ENCODING_INT

REDIS_ENCODING_RAW

Redis 使用 Redis_ENCODING_INT 来存储整数，范围是 32 位有符号整数，即 -2147483648 到 2147483647。

Redis 使用 Redis_RDB_ENC_INT8、Redis_RDB_ENC_INT16 和 Redis_RDB_ENC_INT32 来存储整数。Redis 使用 8 位、16 位和 32 位的整数来存储 integer。

Redis 使用 8 位的整数来存储 123，使用 16 位的整数来存储 10-21。

ENCODING	integer
----------	---------

10-20 INT 类型

REDIS_RDB_ENC_INT8	123
--------------------	-----

10-21 8 位整数

Redis 使用 Redis_ENCODING_RAW 来存储字符串，范围是 0 到 255。

· 字符串长度在 0 到 20 之间，使用 Redis_ENCODING_RAW 存储。

· 字符串长度在 20 以上，使用 Redis_ENCODING_RAW 存储。



Redis 数据库的 RDB 快照文件是二进制格式，不能直接通过文本编辑器查看。RDB 文件是 Redis 数据库的快照，用于备份和恢复。

Redis 的配置文件 redis.conf 中有一个 rdbcompression 选项，用于控制是否对 RDB 文件进行压缩。

Redis 的 RDB 文件默认是压缩的，可以通过配置 redis.conf 中的 rdbcompression 选项来控制。

len	string
-----	--------

图 10-22 Redis RDB 文件的结构

Redis RDB 文件的结构如下：首先是一个 4 字节的 len 字段，表示字符串的长度，然后是字符串本身。RDB 文件是 Redis 数据库的快照，用于备份和恢复。

REDIS_RDB_ENC_LZF	compressed_len	origin_len	compressed_string
-------------------	----------------	------------	-------------------

图 10-23 Redis RDB 文件的结构

Redis 的 RDB 文件默认是压缩的，可以通过配置 redis.conf 中的 rdbcompression 选项来控制。Redis 的 RDB 文件默认是压缩的，可以通过配置 redis.conf 中的 rdbcompression 选项来控制。Redis 的 RDB 文件默认是压缩的，可以通过配置 redis.conf 中的 rdbcompression 选项来控制。

图 10-24 Redis RDB 文件的结构

图10-25 字典的序列化格式。字典的键和值都是字符串，键的长度为21，值为6。字典的键为"?aa???"，值为"?aa???"。

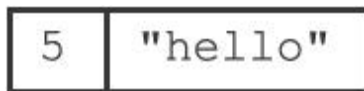


图10-24 字典的键和值

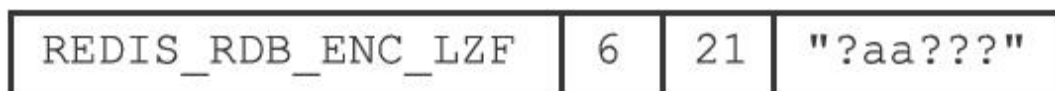


图10-25 字典的序列化格式

2. 列表

列表的TYPE为REDIS_RDB_TYPE_LIST，value为列表。列表的序列化格式为REDIS_ENCODING_LINKEDLIST，RDB的序列化格式为图10-26所示。



图10-26 LINKEDLIST的序列化格式

list_length为列表的长度，item1、item2、...、itemN为列表中的元素。列表的序列化格式为图10-26所示。

每个item的序列化格式为图10-27所示。每个item的序列化格式为图10-27所示。

图10-27 链表结构



图10-27 链表结构

图10-27 链表结构

图

· 链表结构

· 链表结构

· 链表结构

3. 链表

图10-28 链表结构

图10-28 链表结构



图10-28 链表结构

图10-28 链表结构

图

elem 10-29

10-29

4	5	"apple"	6	"banana"	3	"cat"	3	"dog"
---	---	---------	---	----------	---	-------	---	-------

10-29 HT

4

· 5 "apple"

· 6 "banana"

· 3 "cat"

· 3 "dog"

4.

TYPE REDIS_RDB_TYPE_HASH value

REDIS_ENCODING_HT RDB 10-30

· hash_size

·key_value_pair

hash_size	key_value_pair 1	key_value_pair 2	...	key_value_pair N
-----------	------------------	------------------	-----	------------------

10-30 HT

10-31

key1	value1	key2	value2	key3	value3	...
------	--------	------	--------	------	--------	-----

10-31

10-3010-32

hash_size	key1	value1	key2	value2	...	keyN	valueN
-----------	------	--------	------	--------	-----	------	--------

10-32 HT

10-33

2	1	"a"	5	"apple"	1	"b"	6	"banana"
---	---	-----	---	---------	---	-----	---	----------

10-33 HT

2

·1"a"5"apple"

·1"b"6"banana"

5. 字典类型

字典类型（`REDIS_RDB_TYPE_ZSET`）的 value 是有序的。
字典类型（`REDIS_ENCODING_SKIPLIST`）的 RDB 格式如下所示。
10-34

sorted_set_size	element1	element2	...	elementN
-----------------	----------	----------	-----	----------

图 10-34 SKIPLIST 字典的 RDB 格式

sorted_set_size 是字典中元素的数量。
element1, element2, ..., elementN 是字典中的元素。

element 是成员（member）和分数（score）的有序对。
score 是成员（member）的分数（double 类型）。RDB 格式如下所示。
10-35

字典类型（`REDIS_ENCODING_SKIPLIST`）的 RDB 格式如下所示。10-35

member1	score1	member2	score2	member3	score3	...
---------	--------	---------	--------	---------	--------	-----

图 10-35 字典的 RDB 格式

字典类型（`REDIS_ENCODING_SKIPLIST`）的 RDB 格式如下所示。10-36

sorted_set_size	member1	score1	member2	score2	...	memberN	scoreN
-----------------	---------	--------	---------	--------	-----	---------	--------

图 10-36 字典的 RDB 格式

图10-37 字典的存储结构

2	2	"pi"	4	"3.14"	1	"e"	3	"2.7"
---	---	------	---	--------	---	-----	---	-------

图10-37 字典的存储结构

字典的存储结构如图10-37所示，字典的存储结构是一个数组。

·字典的存储结构如图10-37所示，字典的存储结构是一个数组。

·字典的存储结构如图10-37所示，字典的存储结构是一个数组。

·字典的存储结构如图10-37所示，字典的存储结构是一个数组。

·字典的存储结构如图10-37所示，字典的存储结构是一个数组。

6. INTSET

TYPE Redis_RDB_Type_Set_IntSet value

Redis_RDB_Type_Set_IntSet value

Redis_RDB_Type_Set_IntSet

Redis_RDB_Type_Set_IntSet value

Redis_RDB_Type_Set_IntSet value

7. ZIPLIST

TYPE Redis_RDB_Type_List_Ziplist
Redis_RDB_Type_Hash_Ziplist
Redis_RDB_Type_Zset_Ziplist value
RDB

1

2 RDB

RDB TYPE
value

1

2 TYPE TYPE
Redis_RDB_Type_List_Ziplist TYPE
Redis_RDB_Type_Hash_Ziplist TYPE
Redis_RDB_Type_Zset_Ziplist

2 TYPE
RDB

10.4 RDB

RDB Redis RDB
RDB

od Redis RDB
dump -c ASCII -x
od

10.4.1 RDB

RDB

```
redis> FLUSHALL
OK
redis> SAVE
OK
```

od RDB

```
$ od -c dump.rdb
0000000  R E D I S 0 0 0 6 377 334 263 C 360 Z 334
0000020 362 V
0000022
```

```
$ od -c dump.rdb
0000000  R  E  D  I  S  0  0  0  6 376 \0 \0 003  M  S  G
0000020 005  H  E  L  L  O 377 207 z = 304 f  T  L 343
0000037
```

Redis RDB file structure

Header

- Magic number: SELECTDB

- Database number: db_number

- Key-value pairs

od dump.rdb REDIS 0006 376

SELECTDB \0 0 0 0 0

EOF 377 RDB

\0 003 M S G 005 H E L L O

Redis RDB file structure

key value

Header

Redis RDB file structure

TYPE

REDIS_RDB_TYPE_STRING 0 003 MSG

005HELLO

10.4.3 RDB

```
redis> FLUSHALL
OK
redis> SETEX MSG 10086 "HELLO"
OK
redis> SAVE
OK
```

RDB

```
$ od -c dump.rdb
0000000  R  E  D  I  S  0  0  0  6 376 \0 374 \ 2 365 336
0000020  @ 001 \0 \0 \0 003  M  S  G 005  H  E  L  L  O 377
0000040 212 231  x 247 252  } 021 306
0000050
```

·EXPIRETIME_MS

·ms

·TYPE

·keyvalue

0000000000RDB0000000000

·REDIS00060RDB0000000000

·376\000000000000

·374000000EXPIRETIME_MS0

·\2 365 336@001\0\0000000000000000

·\0 003 M S G\0000000000000000003000000MSG0000

·005 H E L L O005000000HELLO0000

·377000EOF0000

·212 231 x 247 252 } 021 3060000000000000

10.4.4 00000000RDB00

0000000000RDB0000000000

```
redis> FLUSHALL
OK
redis> SADD LANG "C" "JAVA" "RUBY"
(integer) 3
redis> SAVE
OK
```

□□□□□□

```
$ od -c dump.rdb
```

0000000 R E D I S 0 0 0 6 376 \0 002 004 L A N

0000020 G 003 004 R U B Y 004 J A V A 001 C 377 202

0000040 312 r 352 346 305 * 023

0000047

□□□RDB□□□□□□□□

```
·REDIS0006 RDB
```

·376\0□□□□0□□□□□

```
·002 004 L A N G 002  REDIS_RDB_TYPE_SET
```

[illegible]

.003

·004 R U B Y□□□□□□□□□□

·004 J A V A□□□□□□□□

.001 C□□□□□□□□

```
·377□□□□□EOF□
```

·202 312 r 352 346 305*023□□□□□□□□

10.4.5 检查RDB文件

Redis提供了RDB检查工具redis-check-dump，用于检查RDB文件是否完整。RDB文件是Redis数据库的快照，如果文件损坏，可能会导致数据丢失。

使用redis-check-dump工具检查RDB文件是否完整。该工具会读取RDB文件并验证其完整性。如果文件损坏，工具会返回错误信息。

使用od命令检查RDB文件的ASCII内容。od命令可以将文件内容以ASCII格式输出，方便查看文件中的文本信息。

使用od命令检查RDB文件的ASCII内容。od命令可以将文件内容以ASCII格式输出，方便查看文件中的文本信息。

```
$ od -cx dump.rdb
0000000 R E D I S 0 0 0 6 377 334 263 C 360 Z 334
          4552 4944 3053 3030 ff36 b3dc f043 dc5a
0000020 362 V
          56f2
0000022
```

使用od命令检查RDB文件的ASCII内容。od命令可以将文件内容以ASCII格式输出，方便查看文件中的文本信息。

使用od命令检查RDB文件的ASCII内容。od命令可以将文件内容以ASCII格式输出，方便查看文件中的文本信息。

10.5 Redis

- RDB Redis

- SAVE

- BGSAVE

- save

BGSAVE

- RDB

- RDB

10.6

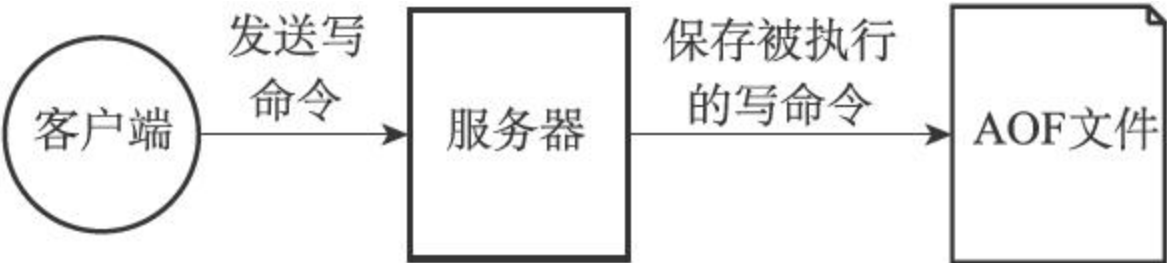
·Sripathi Krishnan Redis RDB RDB RDB RDB / <https://github.com/sripathikrishnan/redis-rdb-tools/wiki/Redis-RDB-Dump-File-Format>

·Sripathi Krishnan Redis RDB RDB Redis 2.6 RDB RDB RDB RDB https://github.com/sripathikrishnan/redis-rdbtools/blob/master/docs/RDB_Version_History.textile

·Redis Redis persistence demystified Redis <http://oldblog.antirez.com/post/redis-persistence-demystified.html> NoSQLFan Redis <http://blog.nosqlfan.com/html/3813.html>

11 AOF

Redis RDB AOF Append Only File
Redis RDB AOF Redis
11-1



□11-1 AOF□□□

[illegible]

```
redis> SET msg "hello"
OK
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> RPUSH numbers 128 256 512
(integer) 3
```

```
RDB[0][0][0][0][0][0][0][0]msgfruitsnumbers[0][0][0][0][0][0][0][0]
RDB[0][0][0]AOF[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]SETSADDERPUSH[0][0][0][0][0][0]
[0][0][0][0]AOF[0][0][0]
```

redis aof redis aof redis
redis aof redis aof redis

redis aof redis aof redis

```
*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n\r\n*3\r\n$3\r\nSET\r\n$3\r\nmsg\r\n$5\r\nhello\r\n\r\n*5\r\n$4\r\nSADD\r\n$6\r\nfruits\r\n$5\r\napple\r\n$6\r\nbanana\r\n$6\r\nche\r\nrry\r\n\r\n*5\r\n$5\r\nRPUSH\r\n$7\r\nnumbers\r\n$3\r\n128\r\n$3\r\n256\r\n$3\r\n512\r\n\r\n\r\n
```

redis aof redis aof redis
redis aof redis aof redis

redis aof redis aof redis
redis aof redis aof redis

```
[8321] 05 Sep 11:58:50.448 # Server started, Redis version 2.9.11  
[8321] 05 Sep 11:58:50.449 * DB loaded from append only file: 0.000  
seconds  
[8321] 05 Sep 11:58:50.449 * The server is now ready to accept connections  
on port 6379
```

redis aof redis aof redis
redis aof redis aof redis

redis aof redis aof redis

11.1 AOF

AOF 通过不断地 `append` 和 `sync` 来记录操作。

11.1.1 配置

AOF 通过 `appendonly` 配置项来启用。默认情况下，Redis 不会启用 AOF。可以通过 `aof_buf` 配置项来指定 AOF 文件的路径。

```
struct redisServer {  
    // ...  
    // AOF  
    sds aof_buf;  
    // ...  
};
```

配置项 `aof_buf` 的默认值是 `appendonly.aof`。

```
redis> SET KEY VALUE  
OK
```

配置项 `appendonly.aof` 的值是 `yes`，表示启用了 AOF。

```
*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n
```

redis> RPUSH NUMBERS ONE TWO THREE

(integer) 3

redis> RPUSH NUMBERS ONE TWO THREE

*5\r\n\$5\r\nRPUSH\r\n\$7\r\nNUMBERS\r\n\$3\r\nONE\r\n\$3\r\nTWO\r\n\$5\r\nTHREE\r\n

redis> AOF

11.1.2 AOF

Redis 的 AOF 持久化机制是 Redis 的三大持久化机制之一，它通过记录 Redis 的写操作来持久化数据。Redis 的 AOF 持久化机制是通过 Redis 的 `serverCron` 函数来实现的。

Redis 的 `serverCron` 函数会调用 `flushAppendOnlyFile` 函数来持久化 AOF 数据。在 `flushAppendOnlyFile` 函数中，Redis 会将 AOF 数据写入到 `aof_buf` 缓冲区中。

```
def eventLoop():
    while True:
        #
        #
        # aof_buf
        #
        processFileEvents()
        #
```



```

    processTimeEvents()
    #
    aof_buf
    AOF
    flushAppendOnlyFile()

```

flushAppendOnlyFile 与 appendfsync 选项

11-1

11-1 flushAppendOnlyFile 与 appendfsync 选项

appendfsync 选项的值	flushAppendOnlyFile 函数的行为
always	将 aof_buf 缓冲区中的所有内容写入并同步到 AOF 文件
everysec	将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，如果上次同步 AOF 文件的时间距离现在超过一秒钟，那么再次对 AOF 文件进行同步，并且这个同步操作是由一个线程专门负责执行的
no	将 aof_buf 缓冲区中的所有内容写入到 AOF 文件，但并不对 AOF 文件进行同步，何时同步由操作系统来决定

appendfsync 选项与 appendfsync 选项

everysec 与 appendfsync 选项 Redis 配置文件

redis.conf

附录

write 函数

Redis 的 write 函数

Redis 的 write 函数

redis数据库操作命令
redis数据库操作命令

redis数据库操作命令
redis数据库操作命令

redis数据库操作命令

-
- 1) SADD databases "Redis" "MongoDB" "MariaDB"
 - 2) SET date "2013-9-5"
 - 3) INCR click_counter 10086
-

redis数据库操作命令

```
*5\r\n$4\r\nSADD\r\n$9\r\ndatabases\r\n$5\r\nRedis\r\n$7\r\nMongoDB\r\n$7\r\nMariaDB\r\n\r\n*3\r\n$3\r\nSET\r\n$4\r\ndate\r\n$8\r\n2013-9-5\r\n\r\n*3\r\n$4\r\nINCR\r\n$13\r\nclick_counter\r\n$5\r\n10086\r\n\r\n
```

redis数据库操作命令
redis数据库操作命令
redis数据库操作命令
redis数据库操作命令

redis数据库操作命令

redis数据库操作命令

appendfsync 0 AOF 0

· appendfsync 1 always 0 aof_buf 0
AOF 0 AOF 1 always 1 appendfsync 0
always 0 AOF 0
0

· appendfsync 2 everysec 0 aof_buf 0
AOF 0 AOF 0
everysec 0

· appendfsync 3 no 0 aof_buf 0
AOF 0 AOF 0 no
flushAppendOnlyFile 0 AOF 0
no everysec 0
no AOF 0

11.2 AOF[1][2][3][4][5][6][7]

Redis AOF [1]
AOF[2]

Redis[3] AOF[4]

1[5] fake client[6] Redis[7]
[8] AOF[9] AOF[10]
[11] AOF[12]
[13]

2[14] AOF[15]

3[16]

4[17] 2[18] 3[19] AOF[20]

[21] AOF[22] 11-2[23]

[24]

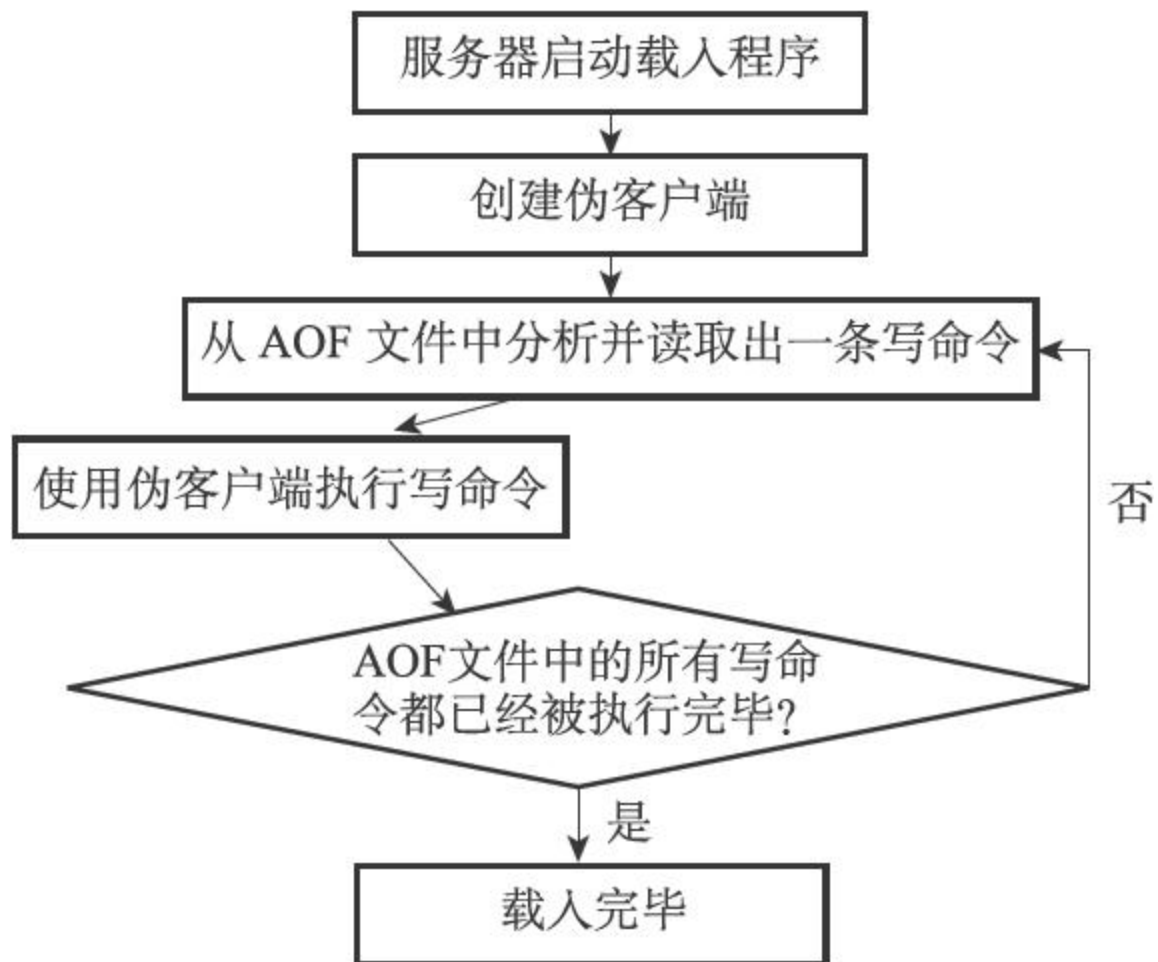


图11-2 AOF加载流程图

下面我们以AOF文件为例

```

*2\r\n$6\r\nSELECT\r\n$1\r\n0\r\n
*3\r\n$3\r\nSET\r\n$3\r\nmsg\r\n$5\r\nhello\r\n
*5\r\n$4\r\nSADD\r\n$6\r\nfruits\r\n$5\r\napple\r\n$6\r\nbanana\r\n$6\r\nche
rry\r\n
*5\r\n$5\r\nRPUSH\r\n$7\r\nnumbers\r\n$3\r\n128\r\n$3\r\n256\r\n$3\r\n512\
r\n
  
```

上面AOF文件中的命令经过Redis的解析，还原成如下命令：

```

SELECT 0
SET msg hello
SADD
fruits apple banana cherry
RPUSH numbers 128 256
  
```

512□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

[illegible]

11.3 AOF

AOF
 AOF
 Redis
 AOF
 AOF
 Redis

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

```
redis> RPUSH list "A" "B"           // ["A", "B"]
(integer) 2
redis> RPUSH list "C"                // ["A", "B", "C"]
(integer) 3
redis> RPUSH list "D" "E"           // ["A", "B", "C", "D", "E"]
(integer) 5
redis> LPOP list                     // ["B", "C", "D", "E"]
"A"
redis> LPOP list                     // ["C", "D", "E"]
"B"
redis> RPUSH list "F" "G"           // ["C", "D", "E", "F", "G"]
(integer) 5
```

```
listAOF
```

[illegible]

```

    AOF
Redis
AOF
rewrite
Redis
AOF
AOF
AOF

```


list AOF
list RPUSH
list "C" "D" "E" "F" "G" AOF list

animals

```
redis> SADD animals "Cat"
// {"Cat"}
(integer) 1
redis> SADD animals "Dog" "Panda" "Tiger" // {"Cat", "Dog", "Panda",
" Tiger"}
(integer) 3
redis> SREM animals "Cat" // {"Dog", "Panda", "Tiger"}
(integer) 1
redis> SADD animals "Lion" "Cat" // {"Dog", "Panda", "Tiger",
" Lion", "Cat"}
(integer) 2
```

animals AOF

animals animals
SADD animals "Dog" "Panda" "Tiger" "Lion" "Cat"
animals

AOF

```

def aof_rewrite(new_aof_file_name):
    #
    """ AOF
    """
    f = create_file(new_aof_file_name)
    #
    """
    for db in redisServer.db:
        #
        """
        if db.is_empty(): continue
        #
        """SELECT
        """
        f.write_command("SELECT" + db.id)
        #
        """
        for key in db:
            #
            """
            if key.is_expired(): continue
            #
            """
            if key.type == String:
                rewrite_string(key)
            elif key.type == List:
                rewrite_list(key)
            elif key.type == Hash:
                rewrite_hash(key)
            elif key.type == Set:
                rewrite_set(key)
            elif key.type == SortedSet:
                rewrite_sorted_set(key)
            #
            """
            if key.have_expire_time():
                rewrite_expire_time(key)

            #
            """
            f.close()
def rewrite_string(key):
    #
    """GET
    """
    value = GET(key)
    #
    """SET

```

```

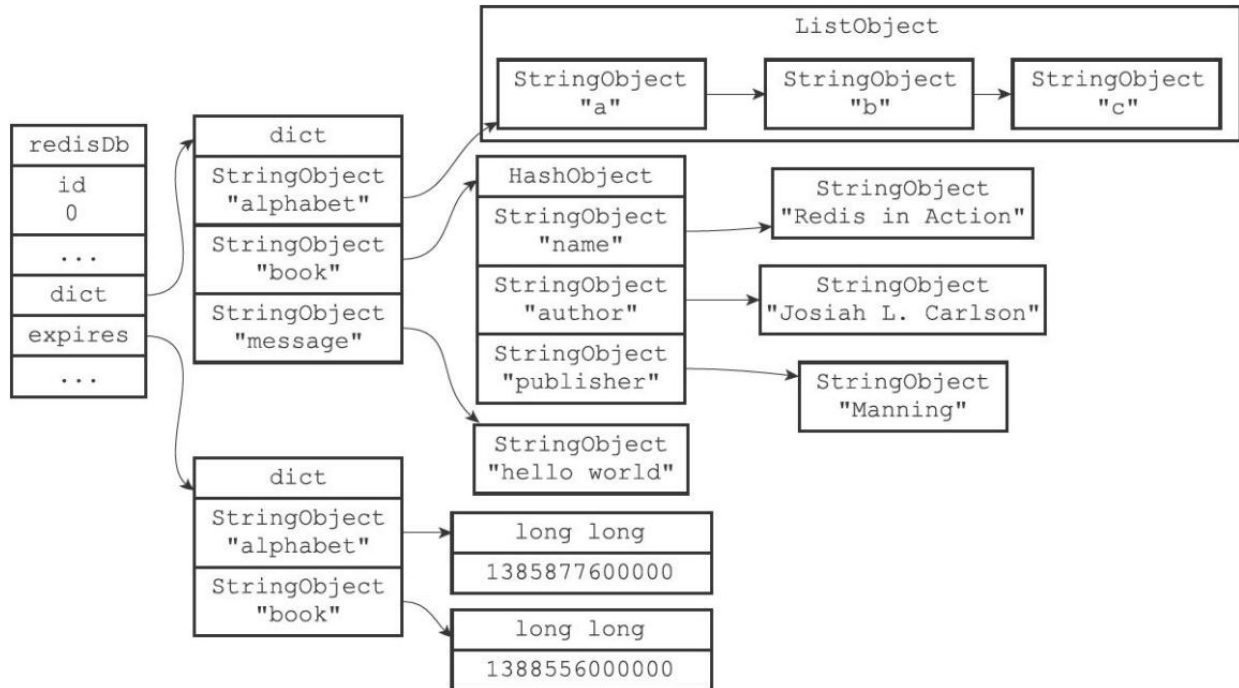
#####
    f.write_command(SET, key, value)
def rewrite_list(key):
    #
    ##LRANGE
    #####
    item1, item2, ..., itemN = LRANGE(key, 0, -1)
    #
    ##RPUSH
    #####
    f.write_command(RPUSH, key, item1, item2, ..., itemN)
def rewrite_hash(key):
    #
    ##HGETALL
    #####
    field1, value1, field2, value2, ..., fieldN, valueN = HGETALL(key)
    #
    ##HMSET
    #####
    f.write_command(HMSET, key, field1, value1, field2, value2, ..., fieldN,
valueN)
def rewrite_set(key):
    #
    ##SMEMBERS
    #####
    elem1, elem2, ..., elemN = SMEMBERS(key)
    #
    ##SADD
    #####
    f.write_command(SADD, key, elem1, elem2, ..., elemN)
def rewrite_sorted_set(key):
    #
    ##ZRange
    #####
    member1, score1, member2, score2, ..., memberN, scoreN = ZRange(key,
0, -1, "WITHSCORES")
    #
    ##ZADD
    #####
    f.write_command(ZADD, key, score1, member1, score2, member2, ...,
scoreN, memberN)
def rewrite_expire_time(key):
    #
    #####
    timestamp = get_expire_time_in_unixstamp(key)
    #
    ##PEXPIREAT

```

redisDb

f.write_command(PEXPIREAT, key, timestamp)

aof_rewrite AOF



11-3

11-3 aof_rewrite AOF

```
SELECT 0
RPUSH alphabet "a" "b" "c"
EXPIREAT alphabet 1385877600000
HMSET book "name" "Redis in Action"
      "author" "Josiah L. Carlson"
      "publisher" "Manning"
EXPIREAT book 1388556000000
SET message "hello world"
```

11-3



redis.h/REDIS_AOF_REWRITE_ITEMS_PER_CMD

REDIS_AOF_REWRITE_ITEMS_PER_CMD 64

64 SADD

64

```
SADD <set-key> <elem1> <elem2> ... <elem64>
SADD <set-key> <elem65> <elem66> ... <elem128>
SADD <set-key> <elem129> <elem130> ... <elem192>
...
```

64 RPUSH

64

```
RPUSH <list-key> <item1> <item2> ... <item64>
RPUSH <list-key> <item65> <item66> ... <item128>
RPUSH <list-key> <item129> <item130> ... <item192>
...
```

11.3.2 AOF

Redis AOF aof_rewrite AOF Redis aof_rewrite AOF

Redis AOF Redis AOF

· AOF

·

AOF AOF

11-2 AOF k1 AOF k2 k3 k4 AOF k1 k1 k2 k3 k4

11-2 AOF

时间	服务器进程	子进程
T1	执行命令 SET k1 v1	
T2	执行命令 SET k1 v2	
T3	执行命令 SET k1 v3	
T4	创建子进程，执行 AOF 文件重写	开始 AOF 文件重写
T5	执行命令 SET k2 10086	执行重写操作
T6	执行命令 SET k3 12345	执行重写操作
T7	执行命令 SET k4 22222	完成 AOF 文件重写

Redis AOF 文件重写

Redis AOF 文件重写

AOF 文件重写 11-4

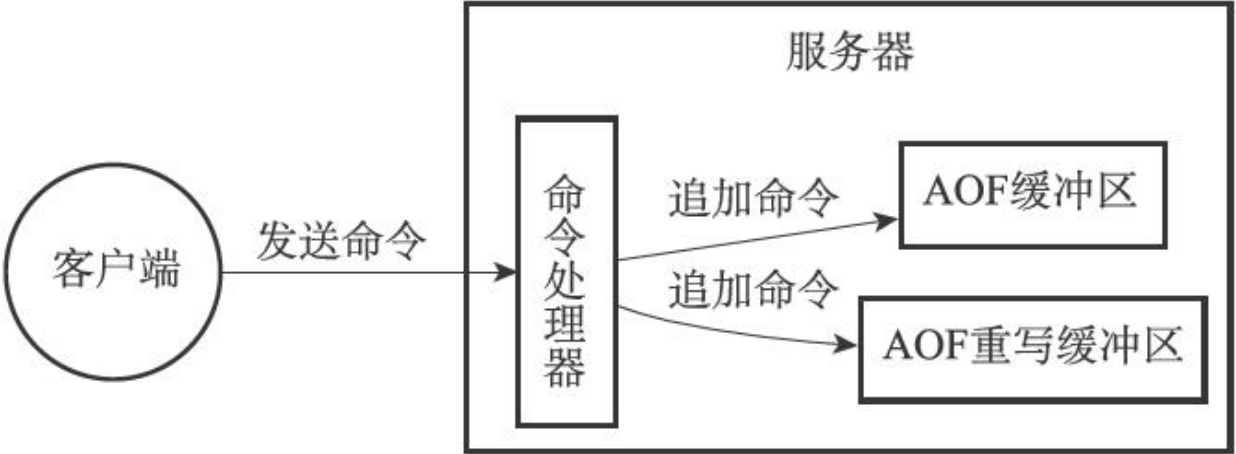


图 11-4 Redis AOF 文件重写

- Redis AOF 文件重写
- Redis AOF 文件重写

3□□□□□□□□□□□□□□□□AOF□□□□□□□□

□ □ □ □ □ □ □ □ □

·AOF□□□□□□□□□□□□□□□□AOF□□□□□□AOF□□□□□□□□□□□□□□□□

AOF

[illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □

1 AOF AOF AOF

□ □ □ □ □ □ □ □ □ □ □ □ □ □

2 AOF atomic AOF AOF

□□□□□

[illegible]

AOFA

[illegible]

11-3 AOF

· k1 AOF

☐☐☐☐☐☐☐☐☐☐☐☐☐☐k2☐☐☐☐k3☐☐☐☐k4☐☐☐☐

· 在 AOF 文件中追加 k2 k3 k4 命令 AOF 文件 AOF 文件 AOF 文件 AOF 文件

11-3 AOF 文件重写

时间	服务器进程（父进程）	子进程
T1	执行命令 SET k1 v1	
T2	执行命令 SET k1 v2	
T3	执行命令 SET k1 v3	
T4	创建子进程，执行 AOF 文件重写	开始 AOF 文件重写
T5	执行命令 SET k2 10086	执行重写操作
T6	执行命令 SET k3 12345	执行重写操作
T7	执行命令 SET k4 22222	完成 AOF 文件重写，向父进程发送信号
T8	接收到子进程发来的信号，将命令 SET k2 10086、SET k3 12345、SET k4 22222 追加到新 AOF 文件的末尾	
T9	用新 AOF 文件覆盖旧 AOF 文件	

在 AOF 文件中追加 BGREWRITEAOF 命令

11.4 配置

- AOF配置

- AOF配置Redis

- AOF配置

- appendfsync配置AOFRedis

配置

- AOF配置

- AOF配置AOF配置AOF配置AOF配置

配置

- AOF配置AOF配置

配置

- BGREWRITEAOF配置RedisAOF配置

配置AOF配置AOF配置AOF配置

配置AOF配置AOF配置AOF配置

配置AOF配置AOF配置AOF配置

12

Redis

·file eventRedisRedis

·time eventRedisserverCron

Redis

API

Redis

12.1 简介

Redis 使用 Reactor 模型实现高性能，其核心组件包括 file event handler

- 使用 epoll 实现 I/O 多路复用 (multiplexing) 以高效处理大量并发连接

- 使用非阻塞的 accept、read、write 和 close 操作，确保在高并发场景下不会阻塞

Redis 使用 Reactor 模型实现高性能，其核心组件包括 Redis 主进程和 Redis 子进程

12.1.1 Redis 主进程

图 12-1 展示了 Redis 主进程的架构，包括 I/O 多路复用器 (dispatcher) 和

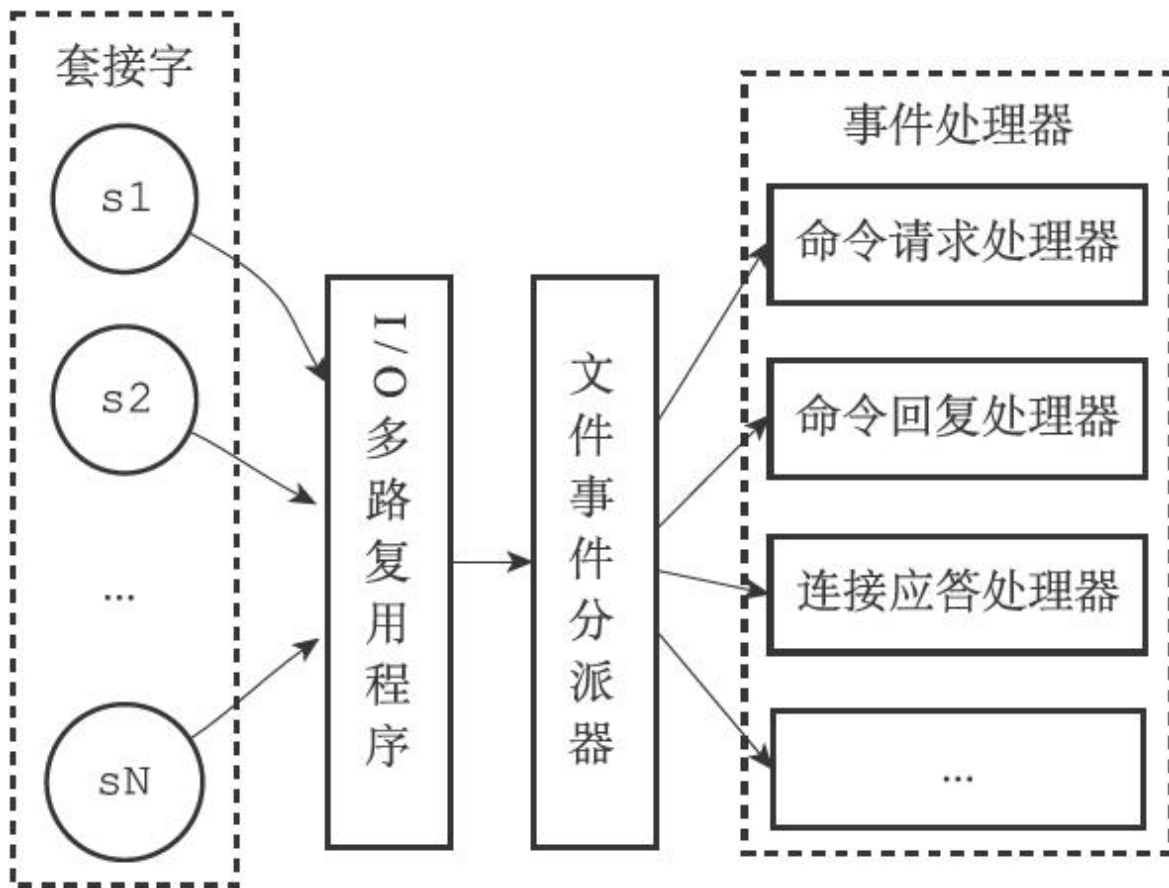


图12-1 网络架构

在Linux中，使用`accept`函数来接受新的连接。该函数会阻塞，直到有新的连接到来。这被称为同步阻塞I/O。在Linux中，使用`select`、`poll`或`epoll`函数来实现I/O多路复用。这些函数会阻塞，直到有数据可读或可写。这被称为同步非阻塞I/O。

I/O模型可以分为同步和异步两种。同步I/O是指，在发送或接收数据之前，必须先等待数据准备好。而异步I/O是指，在发送或接收数据之后，不需要等待数据准备好，就可以继续执行其他操作。在Linux中，使用`read`、`write`、`recv`、`send`等函数来实现同步I/O。而使用`readv`、`writv`、`recvmsg`、`sendmsg`等函数来实现异步I/O。

Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。图 12-2 展示了 Redis 的 I/O 模型。

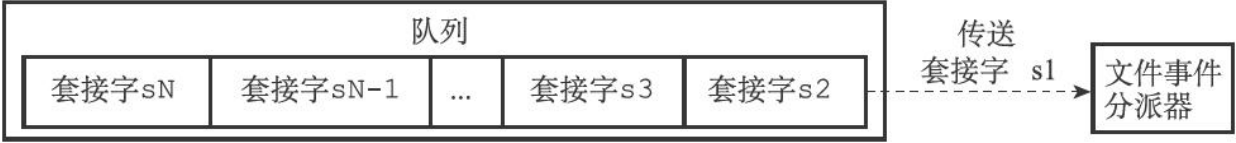


图 12-2 I/O 模型

Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。图 12-2 展示了 Redis 的 I/O 模型。

Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。图 12-2 展示了 Redis 的 I/O 模型。

12.1.2 I/O 模型

Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。Redis 支持多种 I/O 模型，包括 select、epoll、evport 和 kqueue。Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。Redis 支持多种 I/O 模型，包括 select、epoll、evport 和 kqueue。Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。Redis 支持多种 I/O 模型，包括 select、epoll、evport 和 kqueue。

Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。Redis 支持多种 I/O 模型，包括 select、epoll、evport 和 kqueue。Redis 的 I/O 模型是事件驱动模型，其核心组件是文件事件分派器（file event dispatcher）。Redis 支持多种 I/O 模型，包括 select、epoll、evport 和 kqueue。

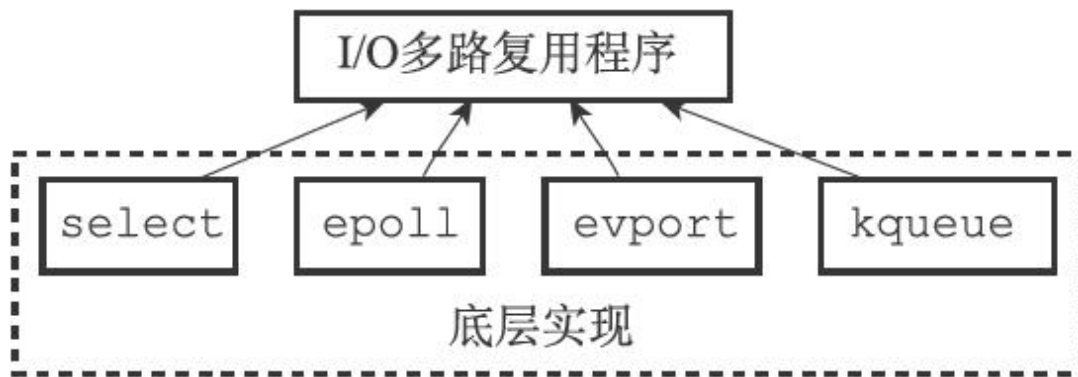


图12-3 Redis的I/O多路复用程序I/O多路复用程序

Redis的I/O多路复用程序使用#include来包含不同的I/O多路复用程序。Redis的I/O多路复用程序使用

```

/* Include the best multiplexing layer supported by this system.
 * The following should be ordered by performances, descending. */
#ifdef HAVE_EVPORT
#include "ae_evport.c"
#else
#ifdef HAVE_EPOLL
#include "ae_epoll.c"
#else
#ifdef HAVE_KQUEUE
#include "ae_kqueue.c"
#else
#include "ae_select.c"
#endif
#endif
#endif
  
```

12.1.3 读写事件

I/O多路复用程序使用ae.h/AE_READABLE和ae.h/AE_WRITABLE来定义读写事件。

· 在写操作完成之前，调用 `write` 函数，然后调用 `close` 函数。
在调用 `accept` 函数之前，调用 `connect` 函数。
`AE_READABLE`

· 在写操作完成之前，调用 `read` 函数。
`AE_WRITABLE`

I/O 操作完成之前，调用 `AE_READABLE` 或 `AE_WRITABLE`。
在调用 `AE_READABLE` 之前，调用 `AE_READABLE`。
在调用 `AE_READABLE` 之前，调用 `AE_WRITABLE`。

在调用 `AE_READABLE` 之前，调用 `AE_READABLE`。

12.1.4 API

`ae.c/aeCreateFileEvent` 函数。
在调用 `aeCreateFileEvent` 之前，调用 `I/O`。

`ae.c/aeDeleteFileEvent` 函数。
在调用 `aeDeleteFileEvent` 之前，调用 `I/O`。

`ae.c/aeGetFileEvents` 函数。

· 在调用 `AE_NONE` 之前，调用 `AE_NONE`。

· 在调用 `AE_READABLE` 之前，调用 `AE_READABLE`。

- 注册可写事件 AE_WRITABLE

- 注册可读事件 AE_READABLE

AE_READABLE|AE_WRITABLE

ae.c/aeWait 阻塞等待事件发生

ae.c/aeApiPoll 非阻塞等待事件发生
sys/time.h/struct timeval 超时时间
aeCreateFileEvent 注册文件事件

ae.c/aeProcessEvents 处理事件

ae.c/aeGetApiName 获取 API 名称
"epoll" 或 "select"

12.1.5 Redis 的 epoll

Redis 使用 epoll 实现高性能

- 注册事件

- 等待事件发生

· 客户端向服务器发起连接请求

· 服务器接收客户端的连接请求

服务器接收客户端的连接请求后，需要为客户端分配一个连接句柄，并启动一个线程来处理客户端的请求。

1. 连接监听

在 `networking.c/acceptTcpHandler` 函数中，Redis 服务器会调用 `sys/socket.h/accept` 函数来监听客户端的连接请求。

Redis 服务器在监听套接字上调用 `sys/socket.h/connect` 函数来监听客户端的连接请求。当有客户端连接时，服务器会产生 `AE_READABLE` 事件，并执行连接应答处理器。图 12-4 展示了连接监听的流程。

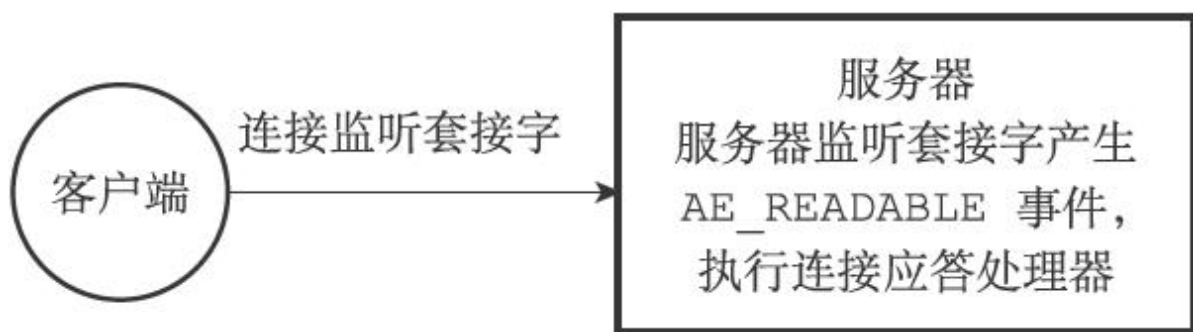


图 12-4 连接监听的流程

2. 连接处理

networking.c/readQueryFromClient Redis
unistd.h/read

AE_READABLE
AE_READABLE 12-5

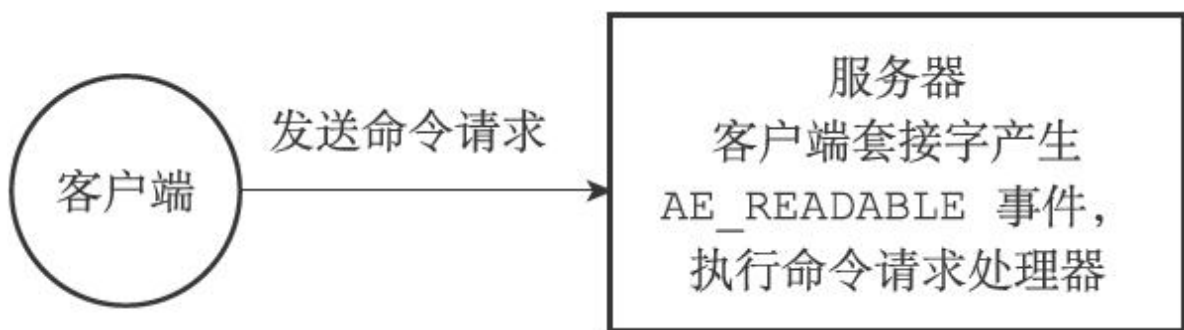


图12-5

AE_READABLE

3.

networking.c/sendReplyToClient Redis
unistd.h/write

AE_WRITABLE
AE_WRITABLE

图12-6 客户端套接字产生AE_WRITABLE事件并发送命令回复



图12-6 客户端套接字产生AE_WRITABLE事件并发送命令回复

当客户端套接字产生AE_WRITABLE事件时，服务器会调用命令回复处理器，将命令回复发送给客户端。

4. 客户端套接字产生AE_READABLE事件

当客户端套接字产生AE_READABLE事件时，服务器会调用命令接收处理器，将命令接收并解析。

当Redis服务器接收到客户端套接字产生的AE_READABLE事件时，服务器会调用命令接收处理器，将命令接收并解析。

当Redis服务器接收到客户端套接字产生的AE_READABLE事件时，服务器会调用命令接收处理器，将命令接收并解析。当命令接收处理器解析完命令后，会调用命令回复处理器，将命令回复发送给客户端。

AE_READABLE
AE_READABLE

AE_WRITABLE
AE_WRITABLE
AE_WRITABLE
AE_WRITABLE

图12-7 客户端与服务器交互过程



图12-7 客户端与服务器交互过程

12.2 Redis

Redis

-

-

- id

- when

- timeProc

-

· 当调用 `AE_NOMORE` 时，表示没有更多的事件需要处理。
 当调用 `when` 时，表示事件将在指定的时间发生。
 当调用 `30` 时，表示事件将在指定的时间发生。
 当调用 `30` 时，表示事件将在指定的时间发生。

Redis 数据库

12.2.1 Redis

Redis 数据库是一个键值对数据库，它支持多种数据类型，如字符串、列表、集合、哈希等。Redis 数据库可以用于缓存、消息队列、分布式数据库等场景。

图 12-8 展示了 Redis 数据库的键值对结构。每个键值对由键（ID）和值（value）组成。图中展示了三个键值对，它们的 ID 分别为 3、2 和 1。每个键值对的值都是一个包含时间事件信息的对象，该对象包含 `time_event`、`id`、`when` 和 `timeProc` 属性。



图 12-8 Redis 数据库的键值对结构

redis 的 ID 的 when
when
redis 的 ID 的 when

Redis serverCron benchmark

Redis serverCron benchmark
benchmark
Redis serverCron benchmark

12.2.2 API

ae.c/aeCreateTimeEvent milliseconds
proc
milliseconds proc

12-9

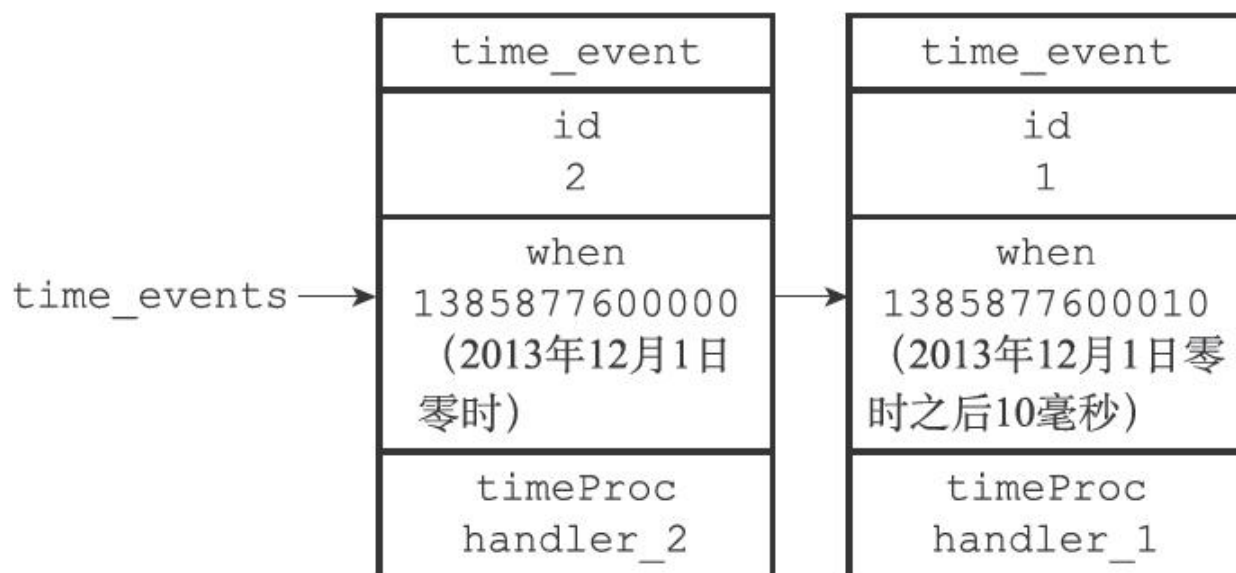


图12-9 时间事件结构

每隔50ms调用handler_3，时间戳为1385877599980。
 2013年12月1日20时，调用aeCreateTimeEvent创建ID为3的时间事件，时间戳为1385877600000，时间间隔为12-8。

调用ae.c/aeDeleteFileEvent删除ID为3的时间事件。
 调用ae.c/aeDeleteFileEvent删除ID为3的时间事件。

每隔50ms调用handler_3，时间戳为1385877599980。
 调用aeDeleteFileEvent删除ID为3的时间事件，时间间隔为12-9。

调用ae.c/aeSearchNearestTimer搜索最近的时间事件。
 每隔50ms调用handler_3，时间戳为1385877599980，2013年12月1日20时，调用aeSearchNearestTimer搜索ID为3的时间事件，时间间隔为12-8。

2

ae.c/processTimeEvents
when
UNIX
UNIX

12-8
1385877600010
2013 12 1
10
processTimeEvents
ID 2
1
1385877600010

processTimeEvents

```
def processTimeEvents():
    #
    for time_event in all_time_event():
        #
        if time_event.when <= unix_ts_now():
            #
            #
            retval = time_event.timeProc()
            #
            if retval == AE_NOMORE:
                #
                delete_time_event_from_server(time_event)
                #
            else:
                #
                when
                #
```

update_when(time_event, retval)

12.2.3 serverCron

Redis serverCron 是 Redis 服务端的一个定时任务，用于执行一些后台任务。它的实现文件是 redis.c/serverCron。

- 定时执行 AOF 持久化
- 定时执行 RDB 持久化
- 定时执行 BGSAVE 持久化
- 定时执行 AOF 重写
- 定时执行 BGREWRITEAOF
- 定时执行 BGSAVE

Redis serverCron 是 Redis 服务端的一个定时任务，用于执行一些后台任务。它的实现文件是 redis.c/serverCron。

Redis 2.6 版本中，serverCron 的默认值是 10 秒，范围是 10 到 100 秒。

12.3 事件驱动模型

事件驱动模型（Event-driven Model）是一种计算机编程模型，它允许程序对事件做出响应。事件可以是用户输入、传感器数据、网络消息等。在事件驱动模型中，程序不会主动去轮询数据，而是等待事件发生后再进行处理。这种模型通常用于处理大量并发连接，如Web服务器、数据库连接池等。

在Python中，事件驱动模型通常通过[asyncio](#)模块来实现。以下是一个简单的示例，展示了如何使用[asyncio](#)模块中的[ae.c/aeProcessEvents](#)函数来处理事件。

```
def aeProcessEvents():
    #
    # 查找下一个定时器事件
    time_event = aeSearchNearestTimer()
    #
    # 计算剩余时间
    remaind_ms = time_event.when - unix_ts_now()
    #
    # 如果剩余时间为负，则设置为0
    if remaind_ms < 0:
        remaind_ms = 0
    #
    # 创建timeval结构体
    timeval = create_timeval_with_ms(remaind_ms)
    #
    # 调用aeApiPoll函数
    aeApiPoll(timeval)
    #
    # 处理文件事件
    processFileEvents()
    #
    # 处理时间事件
    processTimeEvents()
```



12.1 API processFileEvents
aeProcessEvents
processFileEvents

aeProcessEvents Redis

```
def main():  
    #  
    init_server()  
    #  
    while server_is_not_shutdown():  
        aeProcessEvents()  
    #  
    clean_server()
```

Redis 12-10

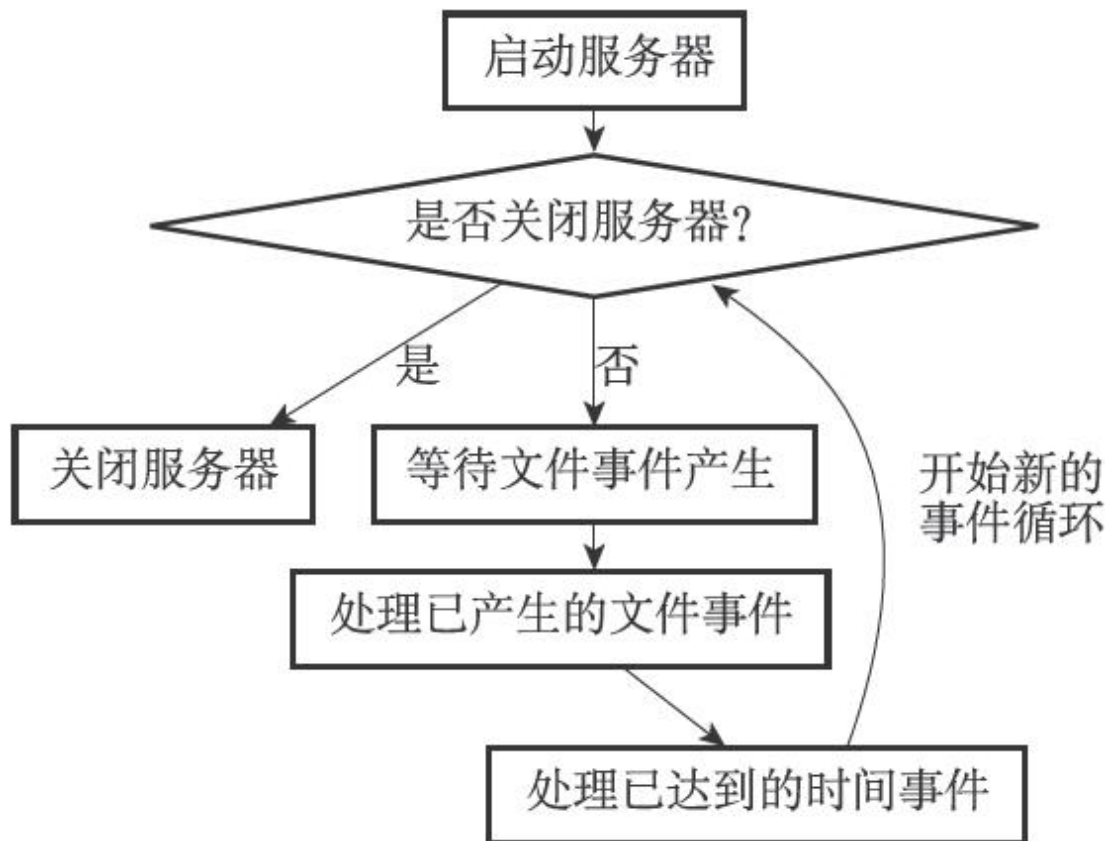


图12-10 服务器操作流程图

服务器操作流程图

1. aeApiPoll 函数用于等待事件的发生，其原型如下：
 int aeApiPoll(aeApiPollCtx *ctx, struct timeval *tv)

2. 该函数的作用是等待事件的发生，并返回事件的类型。其返回值如下：
 - AE_OK: 表示没有事件发生。
 - AE_ERR: 表示发生错误。
 - AE_READABLE: 表示可读事件。
 - AE_WRITEABLE: 表示可写事件。
 - AE_TIME: 表示时间事件。

3. 在循环中，当遇到 break 语句时，循环将终止。break 语句通常用于在循环中找到满足特定条件的元素并立即退出循环。例如，在遍历数组时，一旦找到目标元素，就可以使用 break 来提前结束循环，从而提高效率。

4. 在循环中，当遇到 continue 语句时，循环将跳过当前迭代并直接进入下一次迭代。continue 语句通常用于在遍历集合时跳过不符合条件的元素。例如，在遍历数组时，如果当前元素为 0，可以使用 continue 来跳过该元素并继续处理下一个元素。

表 12-1 循环控制语句

表 12-1 循环控制语句

开始时间	结束时间	动作
0	10	创建一个在 100 毫秒到达的时间事件
11	30	等待文件事件
31	50	处理文件事件
51	85	等待文件事件
85	130	处理文件事件
131	150	执行时间事件

表 12-1 循环控制语句

· 在循环中，当遇到 break 语句时，循环将终止。

12.4 □□□□

- Redis
 - Reactor
 - acceptable
- writable readable
- AE_READABLE AE_WRITABLE
-
- serverCron
-
-
-

12.5 参考

- [Pattern-Oriented Software Architecture](#) [Volume 4: A Pattern Language for Distributed Computing](#) 11月

[Reactor](#) [Reactor](#)

- [Linux System Programming](#) [Second Edition](#) 2月
- [Multiplexed I/O](#) 4月 [Event Poll](#) [Unix](#) 2月
- [14.5](#) [I/O](#)

13 Redis

Redisは、キー・バリュ型データベースです。キーは文字列、バリューは文字列、数値、リスト、ハッシュ、セット、ビットフィールド、地理空間インデックス、JSONなどのデータ型をサポートしています。

Redisは、I/Oマルチプレクシングを採用しています。Redisは、クライアントからの接続を待ち受け、接続が来ると、その接続を処理するためのスレッドを起動します。

Redisは、redis.h/redisClientというヘッダファイルとライブラリを提供しています。このヘッダファイルとライブラリを使用して、Redisクライアントを構築することができます。

- ・Redisクライアントの構造体
- ・Redisクライアントの関数
- ・Redisクライアントのflag
- ・Redisクライアントのオプション
- ・Redisクライアントの接続方法
- ・Redisクライアントの接続オプション
- ・Redisクライアントの接続エラー

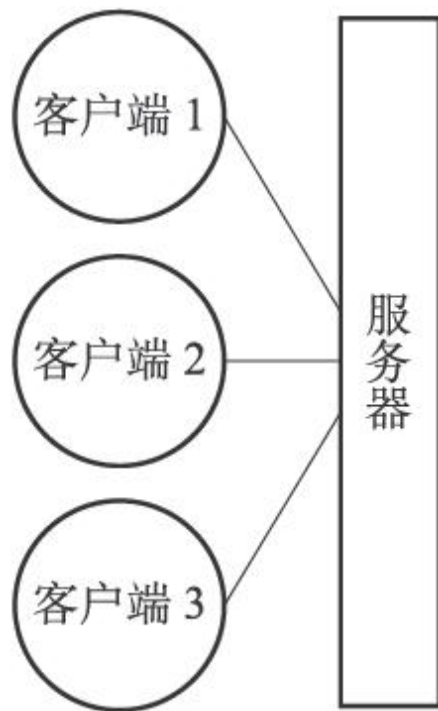


图13-1 客户端连接

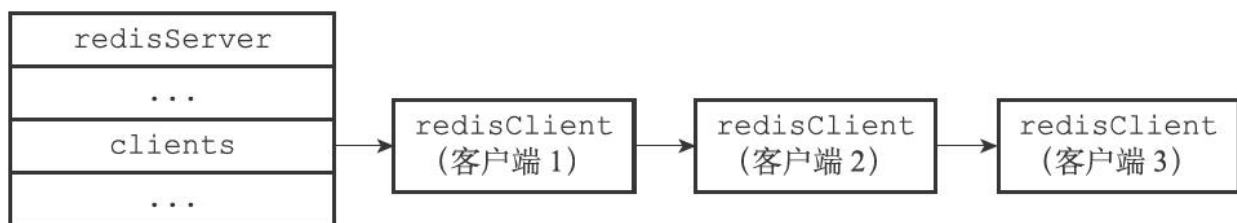


图13-2 clients列表

客户端列表

13.1 redisClient

redisClient结构体定义

· redisClient结构体成员变量

包括

· redisClient结构体成员变量db和dictid

redisClient结构体成员变量mstate和WATCH

redisClient结构体成员变量watched_keys

包括

13.1.1 redisClient

redisClient结构体成员变量fd

```
typedef struct redisClient {  
    // ...  
    int fd;  
    // ...  
} redisClient;
```

redisClient结构体成员变量fd

· fake client fd-1 AOF Redis
Lua Redis
AOF Lua
Redis

· fd-1
fd-1
-1

CLIENT list fd

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name= age=1242 idle=0 ...
addr=127.0.0.1:53469 fd=7 name= age=4 idle=4 ...
```

13.1.2

CLIENT list name

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name= age=1242 idle=0 ...
addr=127.0.0.1:53469 fd=7 name= age=4 idle=4 ...
```

CLIENT setname

CLIENT setname

```
redis> CLIENT list
addr=127.0.0.1:53428 fd=6 name=message_queue age=2093 idle=0 ...
addr=127.0.0.1:53469 fd=7 name=user_relationship age=855 idle=2 ...
```

message_queue

user_relationship

name

```
typedef struct redisClient {
    // ...
    robj *name;
    // ...
} redisClient;
```

name NULL

name

13-3 name

"message_queue"

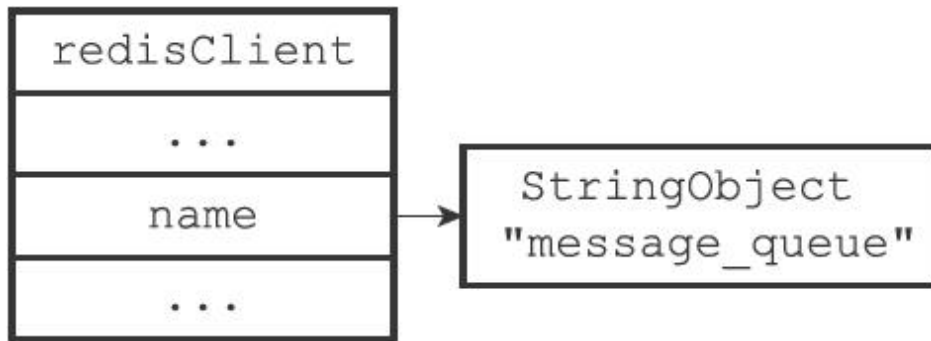


图13-3 name的指向

13.1.3 标志

标志（flags）用于描述客户端的角色（role）和状态（status）。

```

typedef struct redisClient {
    // ...
    int flags;
    // ...
} redisClient;
  
```

标志（flags）的位定义如下：

flags = <flag>

标志（flags）的位定义如下：

flags = <flag1> | <flag2> | ...

标志（flags）的位定义如下：

· 在 Redis 2.8 之前，Redis 的复制模式是单主单从模式，即一个主节点只能有一个从节点，且只能复制主节点的数据。在 Redis 2.8 之后，Redis 引入了 Redis Master 和 Redis Slave 的概念，即一个主节点可以有多个从节点，且每个从节点都可以复制主节点的数据。

· REDIS_PRE_PSYNC 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行全量同步。在 Redis 2.8 之前，主节点和从节点之间只能进行增量同步，即从节点只能复制主节点在复制开始后的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行全量同步，即从节点可以复制主节点在复制开始前的数据。

· REDIS_LUA_CLIENT 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行 Lua 脚本的同步。在 Redis 2.8 之前，主节点和从节点之间只能同步 Lua 脚本的元数据，即脚本的名称和长度。在 Redis 2.8 之后，主节点和从节点之间可以同步 Lua 脚本的元数据和脚本的源代码。

在 Redis 2.8 之前，Redis 的复制模式是单主单从模式，即一个主节点只能有一个从节点，且只能复制主节点的数据。

· REDIS_MONITOR 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行监控。在 Redis 2.8 之前，主节点和从节点之间只能进行数据同步，即从节点只能复制主节点的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行监控，即从节点可以监控主节点的状态。

· REDIS_UNIX_SOCKET 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行 Unix 套接字的同步。在 Redis 2.8 之前，主节点和从节点之间只能进行数据同步，即从节点只能复制主节点的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行 Unix 套接字的同步，即从节点可以复制主节点的 Unix 套接字。

· REDIS_BLOCKED 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行阻塞操作的同步。在 Redis 2.8 之前，主节点和从节点之间只能进行数据同步，即从节点只能复制主节点的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行阻塞操作的同步，即从节点可以复制主节点的阻塞操作。

· REDIS_UNBLOCKED 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行非阻塞操作的同步。在 Redis 2.8 之前，主节点和从节点之间只能进行数据同步，即从节点只能复制主节点的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行非阻塞操作的同步，即从节点可以复制主节点的非阻塞操作。

· REDIS_MULTI 是 Redis 2.8 引入的一个新命令，用于在主节点和从节点之间进行多命令的同步。在 Redis 2.8 之前，主节点和从节点之间只能进行数据同步，即从节点只能复制主节点的数据。在 Redis 2.8 之后，主节点和从节点之间可以进行多命令的同步，即从节点可以复制主节点的多命令。

·REDIS_DIRTY_CAS WATCH
REDIS_DIRTY_EXEC
EXEC
REDIS_MULTI

·REDIS_CLOSE_ASAP
serverCron

·REDIS_CLOSE_AFTER_REPLY CLIENT
KILL

·REDIS Asking ASKING

·REDIS_FORCE_AOF AOF
REDIS_FORCE_REPL PUBSUB
REDIS_FORCE_AOF SCRIPT LOAD
REDIS_FORCE_AOF REDIS_FORCE_REPL

·REPLICATION ACK

REDIS_MASTER_FORCE_REPLY

redis.h

PUBSUB SCRIPT LOAD

Redis AOF
AOF

Redis PUBSUB SCRIPT LOAD
PUBSUB PUBSUB
REDIS_FORCE_AOF AOF AOF
PUBSUB SCRIPT LOAD
PUBSUB SCRIPT LOAD
REDIS_FORCE_AOF
AOF AOF

SCRIPT LOAD
REDIS_FORCE_REPL SCRIPT LOAD

flags

#

```
REDIS_MASTER
#
REDIS_BLOCKED
#
REDIS_MULTI | REDIS_DIRTY_CAS
#
Redis 2.8
REDIS_SLAVE | REDIS_PRE_PSYNC
#
Lua
Redis
#
AOF
REDIS_LUA_CLIENT | REDIS_FORCE_AOF| REDIS_FORCE_REPL
```

13.1.4

```
typedef struct redisClient {
    // ...
    sds querybuf;
    // ...
} redisClient;
```

SET key value

querybufSDS

```
*3\r\n$3\r\nSET\r\n$3\r\nkey\r\n$5\r\nvalue\r\n
```

13-4 SDS querybuf

Redis 2.8.10 版本开始，Redis 客户端使用 SDS 来存储查询缓冲区的字符串，其大小限制为 1GB。

Redis 客户端结构体



13-4 querybuf

13.1.5 客户端

Redis 客户端使用 SDS 来存储查询缓冲区的字符串，其大小限制为 1GB。

Redis 客户端结构体

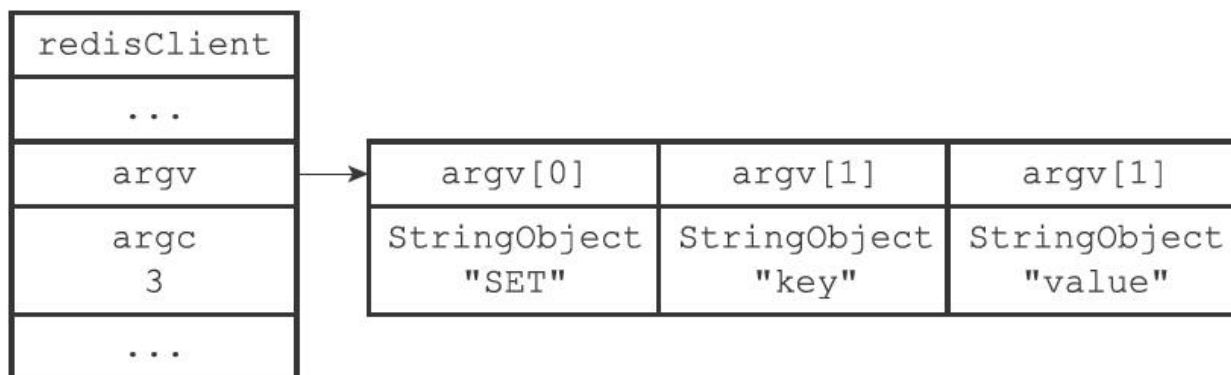
```
typedef struct redisClient {
    // ...
    robj **argv;
    int argc;
    // ...
} redisClient;
```

Redis 客户端使用 SDS 来存储查询缓冲区的字符串，其大小限制为 1GB。

Redis 客户端结构体

```
argc[] [] [] [] [] [] argv[] [] [] [] []
```

```
0000000013-4000querybuf0000000000000000000013-5000argv
000argc000
```



```
13-5   argv[]argc[]
```

```
000013-5000000000argc000030000200000000"SET"0  
00000000
```

13.1.6 〇〇〇〇〇〇〇

```

    argv[0] = "argc";
    argv[1] = "argv[0]";

```

13-6 SDS
redisCommand

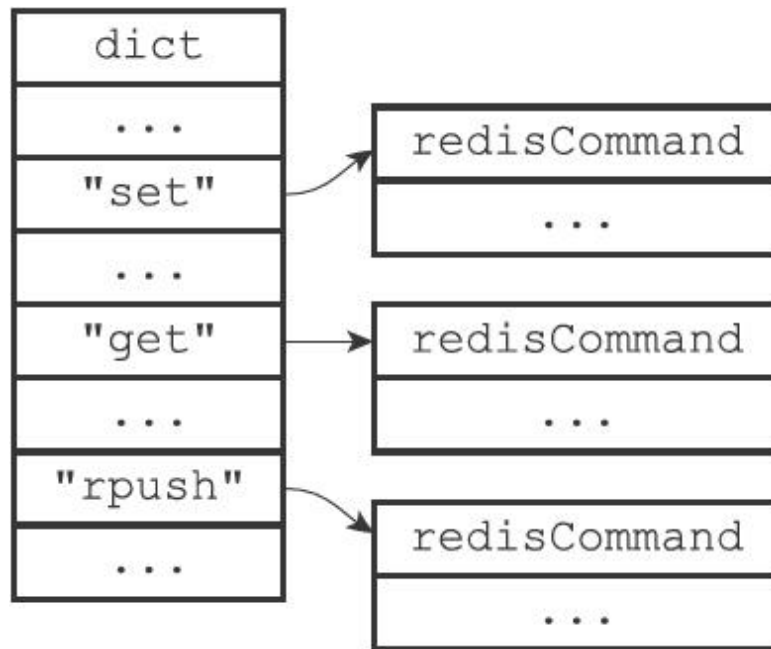


图13-6 字典

字典的键是argv[0]，值是redisCommand结构体。字典的键是命令名，值是命令结构体。

```
typedef struct redisClient {
    // ...
    struct redisCommand *cmd;
    // ...
} redisClient;
```

字典的键是argv[0]，值是redisCommand结构体。字典的键是命令名，值是命令结构体。

图13-7字典的键是argv[0]，值是redisCommand结构体。字典的键是命令名，值是命令结构体。

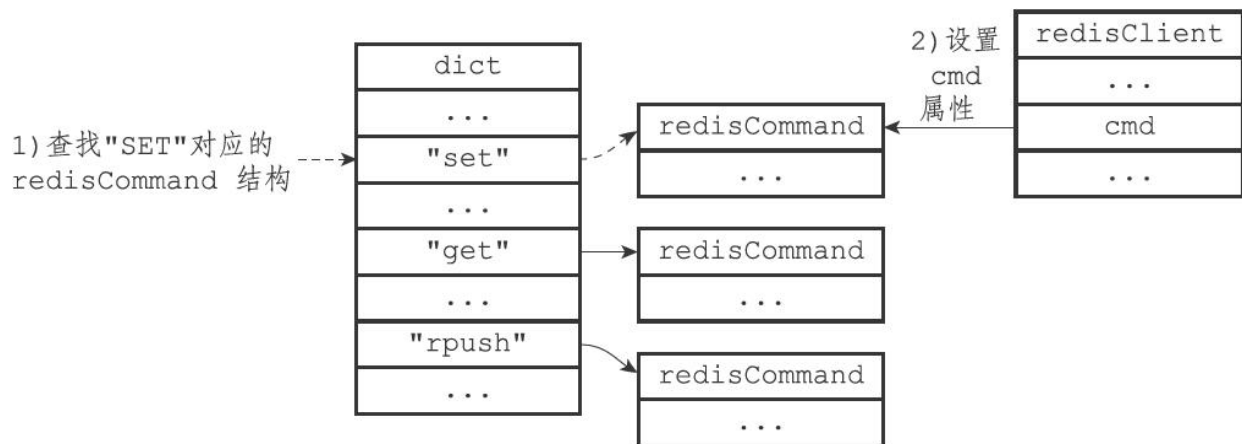


图13-7 查找命令cmd

在main函数中，我们调用redisCommand结构体，将argv[0]的值"SET"和"set"的值
 和"SeT"的值进行比较，如果相等，则返回该结构体。

13.1.7 处理命令

在main函数中，我们调用redisCommand结构体，将argv[0]的值"SET"和"set"的值
 和"SeT"的值进行比较，如果相等，则返回该结构体。

· 如果命令不存在，则返回"OK"，表示命令执行成功。
 · 如果命令存在，则返回该命令的返回值。
 如果命令不存在，则返回"OK"，表示命令执行成功。

在main函数中，我们调用redisCommand结构体，将argv[0]的值"SET"和"set"的值
 和"SeT"的值进行比较，如果相等，则返回该结构体。

```
typedef struct redisClient {
    // ...
    char buf[REDIS_REPLY_CHUNK_BYTES];
    int bufpos;
```

```
// ...
} redisClient;
```

```
buf[bufpos] = REDIS_REPLY_CHUNK_BYTES;
buf[bufpos] = '\0';
```

```
REDIS_REPLY_CHUNK_BYTES = 16 * 1024;
buf[bufpos] = '\0';
```

图 13-8 客户端的 redisClient 结构体

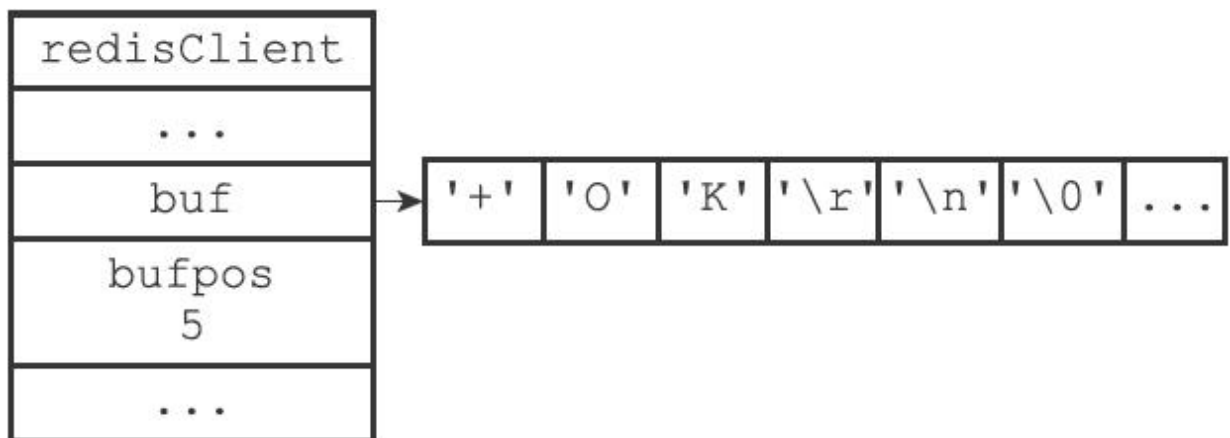


图 13-8 客户端的 redisClient 结构体

```
buf[bufpos] = '\0';
buf[bufpos] = '\0';
```

```
reply[bufpos] = '\0';
```

```
typedef struct redisClient {
    // ...
    list *reply;
```

```
// ...
} redisClient;
```

redisClient->reply 16KB 16KB

图13-9 redisClient->reply 16KB

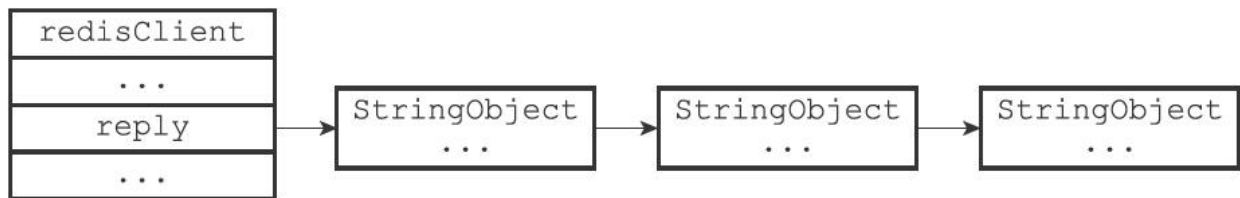


图13-9 redisClient->reply 16KB

13.1.8 redisClient->reply 16KB

redisClient->reply 16KB

```
typedef struct redisClient {
    // ...
    int authenticated;
    // ...
} redisClient;
```

redisClient->reply 16KB

redisClient->reply 16KB

13-10 redisClient->reply 16KB

redisClient
...
authenticated
0
...

图13-10 认证后的客户端

当redisClient的authenticated属性为0时，客户端AUTH命令将失败。

图13-11 认证失败后的客户端

```
redis> PING
(error) NOAUTH Authentication required.
redis> SET msg "hello world"
(error) NOAUTH Authentication required.
```

当客户端AUTH命令成功后，redisClient的authenticated属性将变为1。图13-11展示了认证成功后的客户端。

```
# authenticated
0
1
redis> AUTH 12321
OK
redis> PING
PONG
redis> SET msg "hello world"
OK
```

redisClient
...
authenticated
1
...

图13-11 redisClient结构体

authenticated 表示是否已经认证，默认为0，即未认证。当客户端通过 AUTH 命令认证成功后，authenticated 的值会变为1。如果客户端在认证过程中输入了错误的密码，authenticated 的值会重置为0。

requirepass 表示密码，默认为空字符串。

13.1.9 客户端

Redis 客户端结构体定义如下：

```
typedef struct redisClient {
    // ...
    time_t ctime;
    time_t lastinteraction;
    time_t obuf_soft_limit_reached_time;
    // ...
} redisClient;
```

ctime 表示客户端创建时间，单位为秒。

CLIENT list 命令可以查看当前 Redis 实例中的所有客户端信息。

```
redis> CLIENT list
addr=127.0.0.1:53428 ... age=1242 ...
```

```
lastinteraction[]interaction[]
[]
```

```
lastinteraction[]idle[]
[]CLIENT list[]idle[]
```

```
redis> CLIENT list
addr=127.0.0.1:53428 ... idle=12 ...
```

```
obuf_soft_limit_reached_time[]soft
limit[]
```

13.2 客户端管理

客户端管理是Redis服务器的重要组成部分，本节将详细介绍其实现原理。

13.2.1 客户端连接

当客户端连接到Redis服务器时，服务器会调用`connect`函数进行连接。该函数会检查客户端是否已经存在于`clients`列表中。如果不存在，则会创建一个新的客户端对象并添加到列表中。

例如，当客户端`c1`、`c2`和`c3`连接到服务器时，服务器会将它们添加到`clients`列表中。图13-12展示了客户端在`clients`列表中的存储结构。

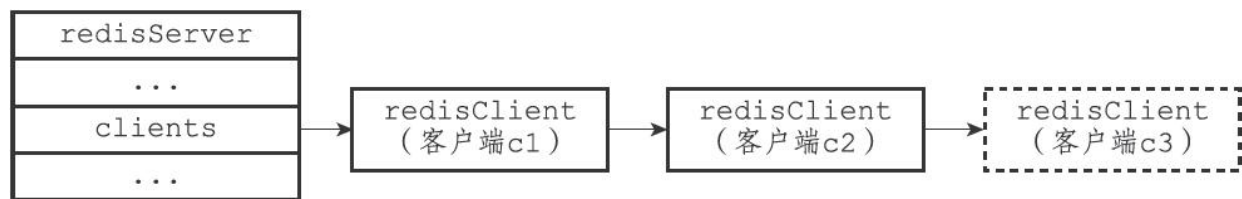


图13-12 客户端在`clients`列表中的存储结构

13.2.2 客户端超时

客户端连接Redis服务器后，如果长时间没有发送任何命令，服务器会认为该客户端已经断开连接。

Redis服务器通过定时扫描`clients`列表来检测超时客户端。如果发现超时客户端，服务器会将其从列表中移除。

[illegible]

· CLIENT KILL

```
· timeout timeout
```

timeout

```
REDIS_MASTER[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] REDIS_SLAVE[ ] [ ] [ ] [ ] BLPOP[ ] [ ]
```

```

00000000 REDIS_BLOCKED00000000 SUBSCRIBE0PSUBSCRIBE

```

timeout

1 GB

□ □ □ □ □ □

[illegible][illegible]

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

[illegible]

□ □

[illegible]

· ☐ hard limit ☐

□ □ □ □ □

· 设置 soft limit 为 0 表示不限制，即当达到软限制时，Redis 不会采取任何措施。
当达到软限制时，Redis 会记录 obuf_soft_limit_reached_time 为当前时间。
当达到软限制时，Redis 会记录 obuf_soft_limit_reached_time 为当前时间。
当达到软限制时，Redis 会记录 obuf_soft_limit_reached_time 为当前时间。
当达到软限制时，Redis 会记录 obuf_soft_limit_reached_time 为当前时间。

设置 client-output-buffer-limit 为 0 0 0 表示不限制。
设置 client-output-buffer-limit 为 256mb 64mb 60 表示当达到硬限制时，Redis 会采取措施。

client-output-buffer-limit <class> <hard limit> <soft limit> <soft seconds>

设置 client-output-buffer-limit 为 0 0 0 表示不限制。

client-output-buffer-limit normal 0 0 0
client-output-buffer-limit slave 256mb 64mb 60
client-output-buffer-limit pubsub 32mb 8mb 60

设置 client-output-buffer-limit 为 0 0 0 表示不限制。
设置 client-output-buffer-limit 为 256MB 64MB 60 表示当达到硬限制时，Redis 会采取措施。
设置 client-output-buffer-limit 为 256MB 64MB 60 表示当达到硬限制时，Redis 会采取措施。

设置 client-output-buffer-limit 为 32MB 8MB 60 表示当达到硬限制时，Redis 会采取措施。
设置 client-output-buffer-limit 为 32MB 8MB 60 表示当达到硬限制时，Redis 会采取措施。

client-output-buffer-limit 配置项在 redis.conf

13.2.3 Lua 扩展

Redis 支持 Lua 脚本扩展，通过 lua_client 结构体

```
struct redisServer {  
    // ...  
    redisClient *lua_client;  
    // ...  
};
```

lua_client 结构体在 redis.h 中定义

13.2.4 AOF 持久化

Redis 支持 AOF 持久化，通过 AOF 持久化 Redis 数据

13.3 〇〇〇〇

- [illegible]

14 项目

Redis 项目
项目地址：[https://github.com/redis/redis](#)

项目地址：[https://github.com/redis/redis](#)
项目地址：[https://github.com/redis/redis](#)

项目地址：[https://github.com/redis/redis](#)
项目地址：[https://github.com/redis/redis](#)

项目地址：[https://github.com/redis/redis](#)
项目地址：[https://github.com/redis/redis](#)

14.1 安装 Redis

Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis 的特点包括高性能、持久化、分布式等。本文将介绍 Redis 的安装和配置。

```
redis> SET KEY VALUE
OK
```

Redis 的 SET 命令用于设置键值对。例如，使用 SET KEY VALUE 命令将键 KEY 的值设置为 VALUE。如果键已经存在，则覆盖其值。返回 OK 表示设置成功。

1. 安装 Redis

SET KEY VALUE

2. 配置 Redis

SET KEY VALUE

OK

3. 启动 Redis

OK

4. 测试 Redis

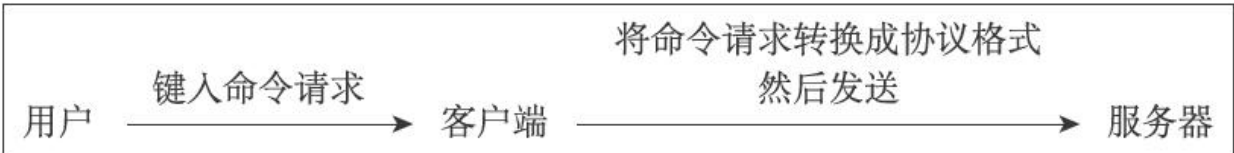
OK

Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表等。Redis 的特点包括高性能、持久化、分布式等。本文将介绍 Redis 的安装和配置。

14.1.1 安装 Redis

Redis 客户端 Redis 服务器

14-1



14-1

SET KEY VALUE

*3\r\n\$3\r\nSET\r\n\$3\r\nKEY\r\n\$5\r\nVALUE\r\n

14.1.2

1

2 在 redisClient 中，argv 和 argc 的值如下：

3 在 redisClient 中，argv 和 argc 的值如下：

在 redisClient 中，SET 命令的 argv 和 argc 的值如下：

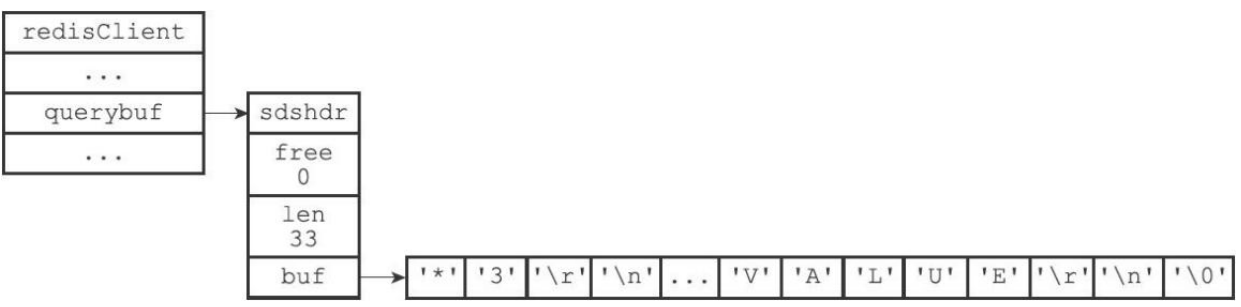


图 14-2 redisClient 的结构

在 redisClient 中，argv 和 argc 的值如下：

*3\r\n3\r\n\r\nSET\r\n\r\n3\r\n\r\nKEY\r\n\r\n5\r\n\r\nVALUE\r\n\r\n\r\n\r\n\r\n0

在 redisClient 中，argv 和 argc 的值如下：图 14-3

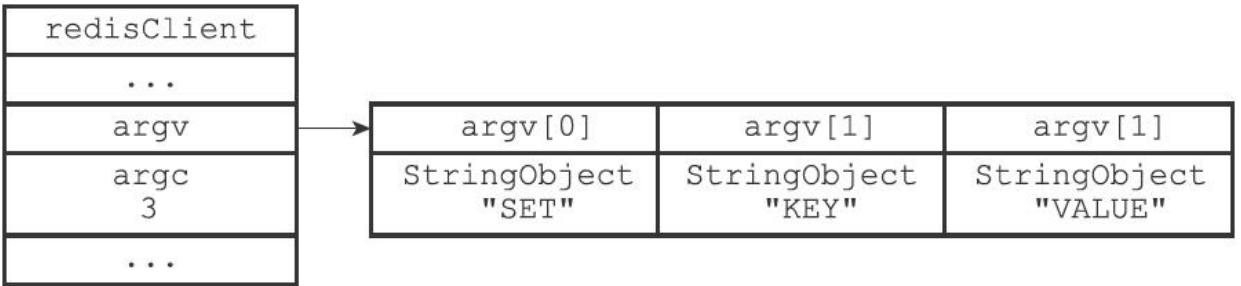


图 14-3 redisClient 的 argv 和 argc

Redis 命令表是 Redis 服务器维护的一个表，它记录了所有 Redis 命令的信息。这个表是一个全局变量，名为 `redisCommandTable`，它包含了一个 `redisCommand` 类型的数组。这个数组的每个元素都是一个 `redisCommand` 结构体，它包含了命令的名称、函数指针、参数个数、标识符、标志位、调用次数和总耗时等信息。

14.1.3 命令表结构

命令表的结构如下所示，它是一个 `redisCommandTable` 类型的变量，它包含了一个 `redisCommand` 类型的数组。这个数组的每个元素都是一个 `redisCommand` 结构体，它包含了命令的名称、函数指针、参数个数、标识符、标志位、调用次数和总耗时等信息。

命令表中的每个元素都是一个 `redisCommand` 结构体，它包含了命令的名称、函数指针、参数个数、标识符、标志位、调用次数和总耗时等信息。这个结构体的定义如下所示：

图 14-1 redisCommand 结构体

属性名	类型	作用
name	char *	命令的名字，比如 "set"
proc	redisCommandProc *	函数指针，指向命令的实现函数，比如 setCommand。redisCommandProc 类型的定义为 typedef void redisCommandProc(redisClient *c);
arity	int	命令参数的个数，用于检查命令请求的格式是否正确。如果这个值为负数 -N，那么表示参数的数量大于等于 N。注意命令的名字本身也是一个参数，比如说 SET msg "hello world" 命令的参数是 "SET"、"msg"、"hello world"，而不仅仅是 "msg" 和 "hello world"
sflags	char *	字符串形式的标识值，这个值记录了命令的属性，比如这个命令是写命令还是读命令，这个命令是否允许在载入数据时使用，这个命令是否允许在 Lua 脚本中使用等等
flags	int	对 sflags 标识进行分析得出的二进制标识，由程序自动生成。服务器对命令标识进行检查时使用的都是 flags 属性而不是 sflags 属性，因为对二进制标识的检查可以方便地通过 &、^、~ 等操作来完成
calls	long long	服务器总共执行了多少次这个命令
milliseconds	long long	服务器执行这个命令所耗费的总时长

图14-2 sflags的含义

图14-2 sflags的含义

标识	意义	带有这个标识的命令
w	这是一个写入命令，可能会修改数据库	SET、RPUSH、DEL 等等
r	这是一个只读命令，不会修改数据库	GET、STRLEN、EXISTS 等等
m	这个命令可能会占用大量内存，执行之前需要先检查服务器的内存使用情况，如果内存紧缺的话就禁止执行这个命令	SET、APPEND、RPUSH、LPUSH、SADD、SINTERSTORE 等等
a	这是一个管理命令	SAVE、BGSAVE、SHUTDOWN 等等
p	这是一个发布与订阅功能方面的命令	PUBLISH、SUBSCRIBE、PUBSUB 等等
s	这个命令不可以在 Lua 脚本中使用	BRPOP、BLPOP、BRPOPLPUSH、SPOP 等等
R	这是一个随机命令，对于相同的数据集和相同的参数，命令返回的结果可能不同	SPOP、SRANDMEMBER、SSCAN、RANDOMKEY 等等
S	当在 Lua 脚本中使用这个命令时，对这个命令的输出结果进行一次排序，使得命令的结果有序	SINTER、SUNION、SDIFF、SMEMBERS、KEYS 等等
l	这个命令可以在服务器载入数据的过程中使用	INFO、SHUTDOWN、PUBLISH 等等
t	这是一个允许从服务器在带有过期数据时使用的命令	SLAVEOF、PING、INFO 等等
M	这个命令在监视器（monitor）模式下不会自动被传播（propagate）	EXEC

图14-4 使用SET和GET命令

redisCommand

```
·SET "set" setCommand -3
```

```
·····"wm" SET
```

redisCommand *redisCommandTable;

·GET 命令的 name 为 "get"，proc 为 getCommand，arity 为 2，sflags 为 "r"。

SET 命令的 name 为 "set"，proc 为 setCommand，arity 为 3，sflags 为 "wm"。

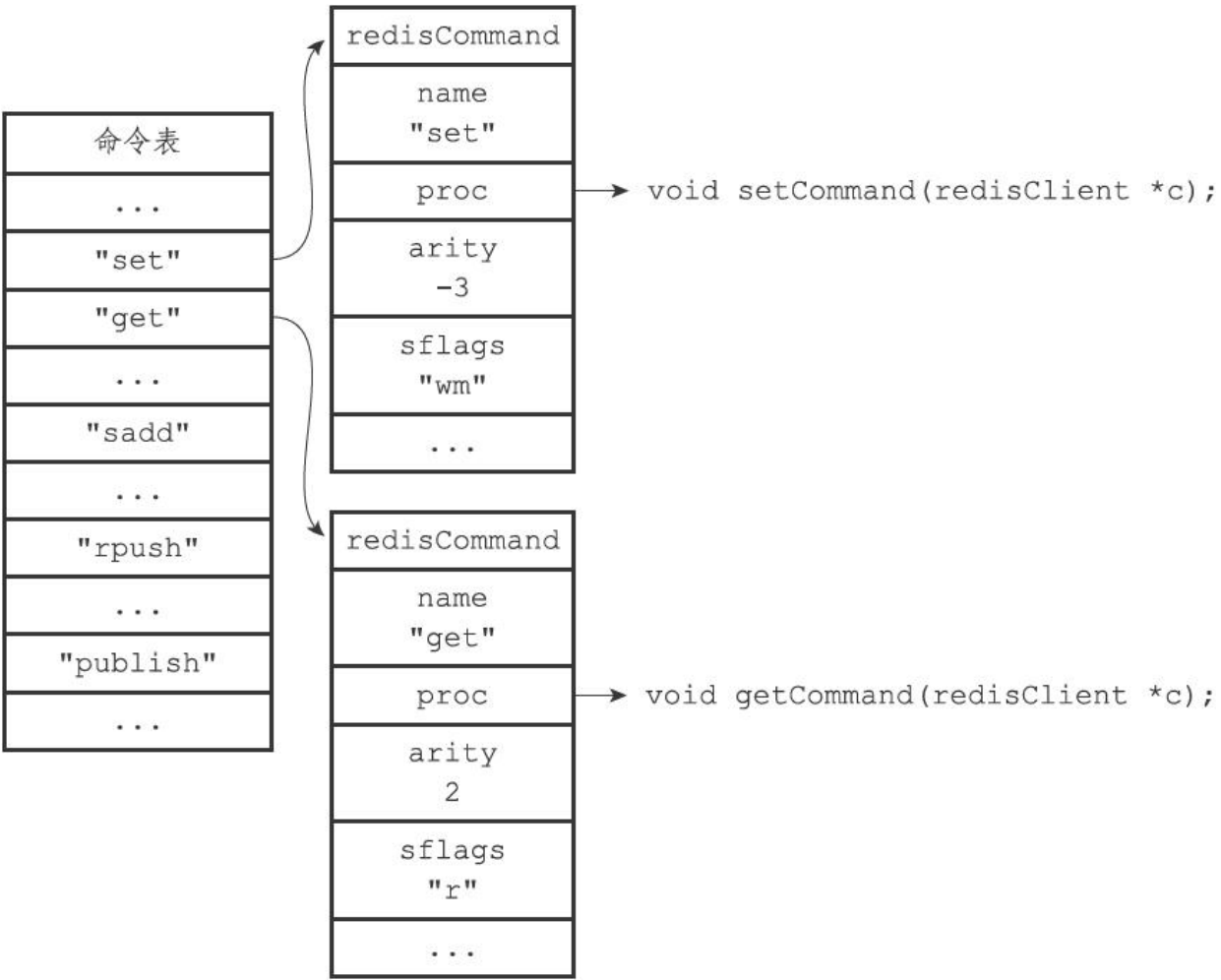


图 14-4 Redis 命令表


```
redis> sEt msg "hello world"
OK
```

14.1.4 Redis 2.x 版本

Redis 2.x 版本中，`cmd` 成员变量用于记录当前命令的名称，`argv` 成员变量用于记录当前命令的参数字符串数组，`argc` 成员变量用于记录当前命令的参数字符串数组的大小。

· 当命令名称为 `cmd` 成员变量为 `NULL` 时，表示当前命令正在执行，此时不能对 `cmd` 成员变量进行赋值。

· 当命令名称为 `cmd` 成员变量为 `redisCommand` 成员变量为 `arity` 成员变量为 `arity` 时，表示当前命令正在执行，此时不能对 `cmd` 成员变量进行赋值。当 `redisCommand` 成员变量为 `arity` 成员变量为 `-3` 时，表示当前命令正在执行，此时不能对 `cmd` 成员变量进行赋值。

· 当命令名称为 `cmd` 成员变量为 `AUTH` 时，表示当前命令正在执行，此时不能对 `cmd` 成员变量进行赋值。

· 当命令名称为 `cmd` 成员变量为 `maxmemory` 时，表示当前命令正在执行，此时不能对 `cmd` 成员变量进行赋值。

· 在 Redis 中，BGSAVE 命令会阻塞主进程，直到所有数据都写入磁盘。
stop-writes-on-bgsave-error 配置项可以设置当 BGSAVE 失败时，是否停止写入。
默认情况下，Redis 会在主进程空闲时执行 BGSAVE。

· 在 Redis 中，SUBSCRIBE 命令用于订阅一个或多个频道，PUBLISH 命令用于向一个或多个频道发布消息。
UNSUBSCRIBE 命令用于取消订阅，PUNSUBSCRIBE 命令用于取消所有订阅。

· 在 Redis 中，INFO 命令用于获取 Redis 服务器的各种信息。
SHUTDOWN 命令用于关闭 Redis 服务器，PUBLISH 命令用于向一个或多个频道发布消息。

· 在 Redis 中，Lua 脚本可以用于执行一系列 Redis 命令。
SHUTDOWN nosave 命令用于关闭 Redis 服务器，而不保存数据。
SCRIPT KILL 命令用于杀死正在执行的 Lua 脚本。

· 在 Redis 中，EXEC 命令用于执行一个 Lua 脚本，DISCARD 命令用于取消一个 Lua 脚本。
MULTI 命令用于开始一个 Lua 脚本，WATCH 命令用于监视一个或多个键。

· 在 Redis 中，CONFIG 命令用于配置 Redis 服务器的各种参数。
SET 命令用于设置一个键的值，GET 命令用于获取一个键的值。



在 Redis 中，SET 命令用于设置一个键的值，GET 命令用于获取一个键的值。
SET 命令的语法是 SET key value，GET 命令的语法是 GET key。

14.1.5 3

cmd
argv argv

```
// client  
client->cmd->proc(client);
```

argv

SET 14-6

```
client->cmd->proc(client);
```

```
setCommand(client);
```

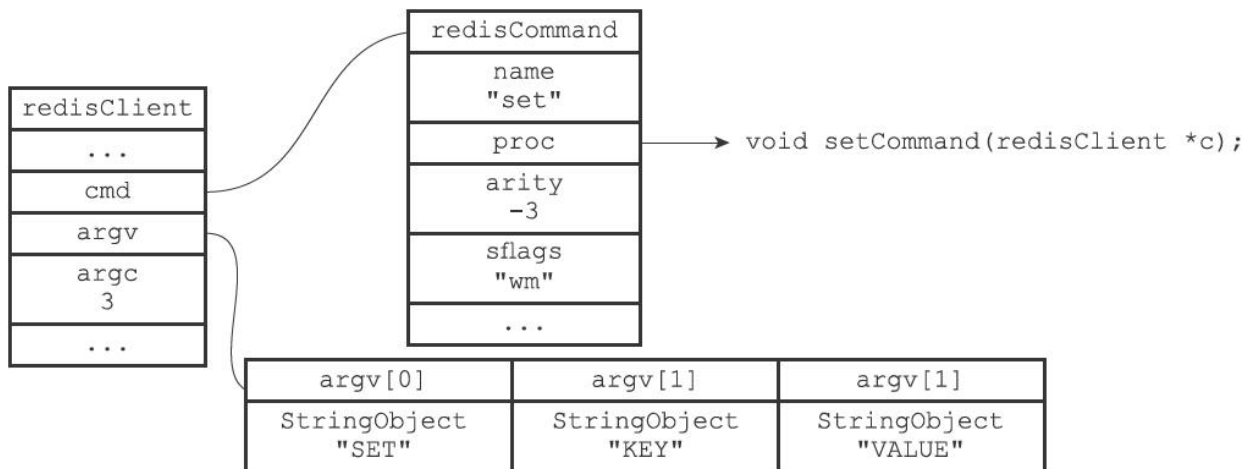


图14-6 命令结构

在 `setCommand` 函数中，首先检查命令是否有效，然后检查命令的格式是否正确。如果命令格式正确，则将命令的返回值写入 `buf`，并调用 `reply` 函数将结果返回给客户端。

在 `SET` 命令的实现中，调用 `setCommand` 函数，并传入客户端的 `client` 指针。最后，将 `"OK\r\n"` 写入 `buf`，并调用 `reply` 函数将结果返回给客户端。

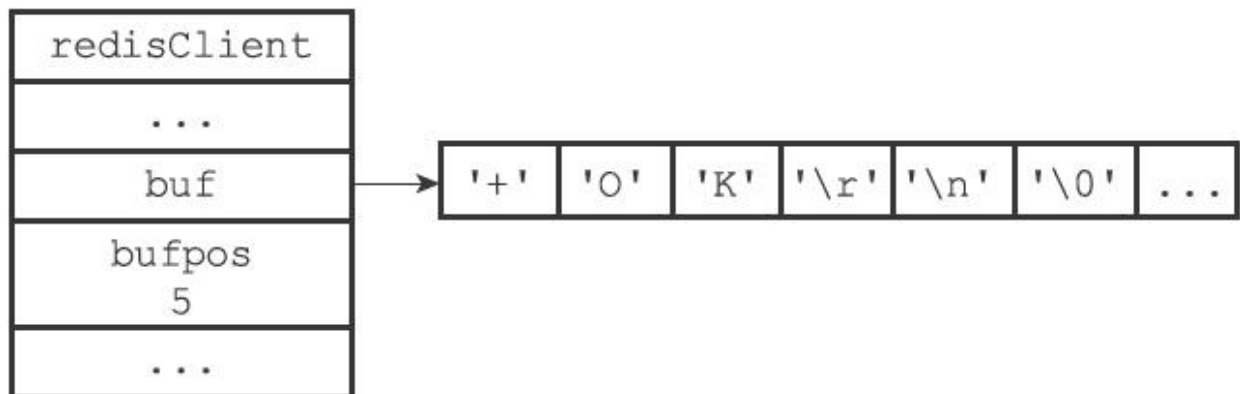


图14-7 命令返回值

14.1.6 命令格式4

redisCommand() 函数

· redisCommand() 函数返回一个字符串，表示命令执行的结果。
如果命令执行失败，则返回 NULL。

· redisCommand() 函数的第二个参数是命令的超时时间，单位为
milliseconds。如果命令执行时间超过这个时间，则返回 NULL。

· redisCommand() 函数的第三个参数是命令的回调函数，用于处理
命令执行的结果。

· redisCommand() 函数的第四个参数是命令的上下文指针，用于
在回调函数中访问上下文信息。

redisCommand() 函数返回的字符串，表示命令执行的结果。
如果命令执行失败，则返回 NULL。

14.1.7 异步命令

Redis 支持异步命令，即可以在主线程中执行命令，而不需要
等待命令执行完成。异步命令的执行结果会在主线程中异步返回。
异步命令的执行结果可以通过 redisCommandAsync() 函数获取。

redisCommandAsync() 函数

图 14-7 异步命令的执行过程
redisCommandAsync() 函数返回一个字符串，表示命令执行的结果。
如果命令执行失败，则返回 NULL。

14.1.8 返回给客户端

Redis 的回复处理器将协议格式的回复返回给客户端

Redis 的 redis-cli 客户端 14-8

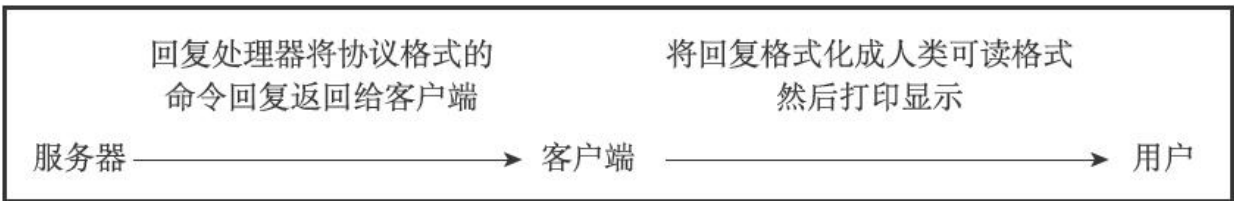


图 14-8 Redis 的回复处理器

Redis 的 SET 命令返回给客户端 "+OK\r\n"

Redis 的 SET 命令返回给客户端 "+OK\r\n"

```
redis> SET KEY VALUE
OK
```

Redis 的 SET 命令返回给客户端 "+OK\r\n"

14.2 serverCron

Redis 的 `serverCron` 函数每隔 100 毫秒执行一次，用于检查服务器的状态。

该函数会遍历所有的 `serverCron` 函数，并调用 `redisServer` 函数来检查服务器的状态。

14.2.1 检查服务器的状态

Redis 的 `redisServer` 函数会检查服务器的状态，包括 `unixtime` 和 `mstime` 等。

```
struct redisServer {  
    // ...  
    //  
    time_t unixtime;  
    //  
    long long mstime;  
    // ...  
};
```

serverCron 100 unixtime mstime

· LRU uptime

·

14.2.2 LRU

lruclk LRU unixtime mstime

```
struct redisServer {  
    // ...  
    //  
    10  
    //  
    idle  
    //  
    unsigned lruclk:22;  
    // ...  
};
```

Redis lru lru

```
typedef struct redisObject {  
    // ...  
    unsigned lru:22;
```

```
// ...  
} robj;
```

redis 2.8.10 64bit
redis 2.8.10 64bit

lruclock redis 2.8.10 64bit

```
redis> SET msg "hello world"  
OK  
#  
redis> OBJECT IDLETIME msg  
(integer)20  
#  
redis> OBJECT IDLETIME msg  
(integer)180  
#  
redis> GET msg  
"hello world"  
#  
redis> OBJECT IDLETIME msg  
(integer)0
```

serverCron redis 2.8.10 64bit

LRU redis 2.8.10 64bit

lruclock redis 2.8.10 64bit

```
redis> INFO server  
# Server  
...  
lru_clock:55923  
...
```

14.2.3 跟踪每秒操作数

serverCron 跟踪每秒操作数，每 100 毫秒采样一次，并更新 `instantaneous_ops_per_sec`。

```
redis> INFO stats
# Stats
...
instantaneous_ops_per_sec:6
...
```

跟踪每秒操作数，每 100 毫秒采样一次，并更新 `instantaneous_ops_per_sec`。

跟踪每秒操作数，每 100 毫秒采样一次，并更新 `instantaneous_ops_per_sec`。

```
struct redisServer {
    // ...
    //
    long long ops_sec_last_sample_time;
    //
    long long ops_sec_last_sample_ops;
    // REDIS_OPS_SEC_SAMPLES
    16
    //
    long long ops_sec_samples[REDIS_OPS_SEC_SAMPLES];
    // ops_sec_samples
    //
    //
}
```

```

//
REDIS_16
REDIS_0
{
//
ops_sec_samples
REDIS_
int ops_sec_idx;
// ...
};

```

```

trackOperationsPerSecondREDIS_
ops_sec_last_sample_timeREDIS_
ops_sec_last_sample_opsREDIS_
REDIS_ trackOperationsPerSecondREDIS_
REDIS_ 1000REDIS_
REDIS_ ops_sec_samplesREDIS_

```

```

REDIS_ INFOREDIS_ getOperationsPerSecondREDIS_
ops_sec_samplesREDIS_ instantaneous_ops_per_sec
REDIS_ getOperationsPerSecondREDIS_

```

```

long long getOperationsPerSecond(void){
    int j;
    long long sum = 0;
    //
    REDIS_
    for (j = 0; j < REDIS_OPS_SEC_SAMPLES; j++)
        sum += server.ops_sec_samples[j];
    //
    REDIS_
    return sum / REDIS_OPS_SEC_SAMPLES;
}

```

```
    opsPerSecondOpsPerSecond
instantaneous_ops_per_secinstantaneous_ops_per_sec
REDIS_OPS_SEC_SAMPLESREDIS_OPS_SEC_SAMPLES
```

14.2.4 内存峰值统计

Redis 使用 `stat_peak_memory` 来统计内存峰值。

```
struct redisServer {
    // ...
    //
    size_t stat_peak_memory;
    // ...
};
```

```
serverCronserverCron
stat_peak_memorystat_peak_memory
stat_peak_memorystat_peak_memory
stat_peak_memorystat_peak_memory
```

```
INFO memoryused_memory_peak
used_memory_peak_humanused_memory_peak_human
```

```
redis> INFO memory
# Memory
...
used_memory_peak:501824
used_memory_peak_human:490.06K
...
```

14.2.5 SIGTERM

Redis 注册 SIGTERM 信号处理器 sigtermHandler，当收到 SIGTERM 信号时，设置 shutdown_asap 为 1。

```
// SIGTERM
// 注册信号处理器
static void sigtermHandler(int sig) {
    //
    // 记录日志
    redisLogFromHandler(REDIS_WARNING, "Received SIGTERM, scheduling shutdown...");
    //
    // 设置 shutdown_asap 为 1
    server.shutdown_asap = 1;
}
```

serverCron 函数会定期检查 shutdown_asap 是否为 1，如果是，则开始关闭 Redis。

```
struct redisServer {
    // ...
    //
    // 是否正在关闭
    int shutdown_asap;
    //
    // 是否正在关闭
    int shutdown_asap;
    // ...
};
```

redis-cli SIGTERM redis-cli

```
[6794 | signal handler] (1384435690) Received SIGTERM, scheduling shutdown...
[6794] 14 Nov 21:28:10.108 # User requested shutdown...
[6794] 14 Nov 21:28:10.108 * Saving the final RDB snapshot before exiting.
[6794] 14 Nov 21:28:10.161 * DB saved on disk
[6794] 14 Nov 21:28:10.161 # Redis is now ready to exit, bye bye...
```

redis-cli RDB redis-cli SIGTERM
redis-cli SIGTERM redis-cli

14.2.6 客户端

serverCron redis-cli clientsCron redis-cli clientsCron redis-cli
redis-cli

· redis-cli
redis-cli

· redis-cli
redis-cli

14.2.7 数据库

serverCron redis-cli databasesCron redis-cli
redis-cli 9 redis-cli

□□

14.2.8 □□□□□BGREWRITEAOF

□□□□□BGSAVE□□□□□□□□□□□□□□□BGREWRITEAOF□□□□□
□□□□□BGREWRITEAOF□□□□□□□□□□BGSAVE□□□□□□□□□□

□□□□aof_rewrite_scheduled□□□□□□□□□□□□□□□
BGREWRITEAOF□□□□

```
struct redisServer {  
    // ...  
    //  
    □□□□1  
    □□□□□ BGREWRITEAOF  
    □□□□□□□  
    int aof_rewrite_scheduled;  
    // ...  
};
```

□□serverCron□□□□□□□□□□□□□□□BGSAVE□□□□□BGREWRITEAOF
□□□□□□□□□□□□□□□□□□□□□□□□□□□□aof_rewrite_scheduled□□□□□1□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□BGREWRITEAOF□□□□

14.2.9 □□□□□□□□□□□□□□□

□□□□□□□rdb_child_pid□□□□aof_child_pid□□□□□□□□□□BGSAVE□□□□
BGREWRITEAOF□□□□□□□□□□ID□□□□□□□□□□□□□□□□□□□□□□BGSAVE□□□□□

BGREWRITEAOF

```
struct redisServer {
    // ...
    //
    int BGSAVE
    int ID
}
//
int BGSAVE
//
int -1
pid_t rdb_child_pid;          /* PID of RDB saving child */
//
int BGREWRITEAOF
int ID
//
int BGREWRITEAOF
//
int -1
pid_t aof_child_pid;          /* PID if rewriting process */
// ...
};
```

serverCron() 函数中，当 rdb_child_pid 不为 0 且 aof_child_pid 不为 0 时，会调用 wait3() 函数等待子进程结束。如果子进程在 1 秒内没有结束，则会调用 wait3() 函数等待子进程结束。如果子进程在 1 秒内没有结束，则会调用 wait3() 函数等待子进程结束。

· 当 Redis 正在执行 BGSAVE 或 BGREWRITEAOF 时，主进程会调用 fork() 函数创建子进程。子进程会执行 BGSAVE 或 BGREWRITEAOF 操作，而主进程会继续处理客户端请求。当子进程完成操作后，主进程会调用 wait3() 函数等待子进程结束。如果子进程在 1 秒内没有结束，则会调用 wait3() 函数等待子进程结束。如果子进程在 1 秒内没有结束，则会调用 wait3() 函数等待子进程结束。

·

rdb_child_pid aof_child_pid-1

1 BGREWRITEAOF

2 BGSAVE 1 BGREWRITEAOF

3 AOF 1 2

14-9

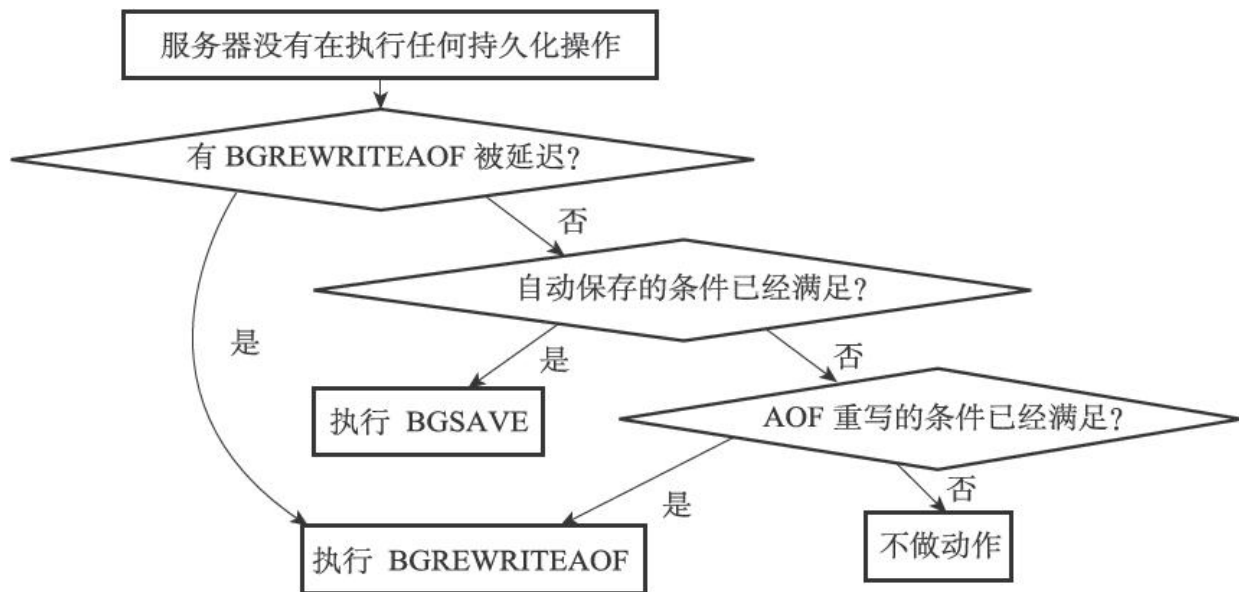


图 14-9 Redis 持久化流程图

14.2.10 Redis AOF 持久化与 AOF 重写

Redis 的 AOF 持久化与 AOF 重写是 Redis 持久化的一部分，由 `serverCron` 函数负责。AOF 持久化与 AOF 重写的频率由 `11` 个配置项控制。

14.2.11 Redis 配置项

Redis 的 AOF 持久化与 AOF 重写的配置项如下表 13 所示。

14.2.12 Redis `cronloops` 配置项

Redis 的 `cronloops` 配置项与 `serverCron` 配置项的关系如下。

```

struct redisServer {
    // ...
    // serverCron

```

```
int main()
{
    // serverCron
    int cronloops;
    // ...
};
```

cronloops = serverCron();

N = ...;

```
if (cronloops % N == 0)
{
    #
    ...
}
```

14.3 初始化

Redis 的初始化过程主要包含以下几个步骤：

1. 初始化全局变量。
2. 初始化 Redis 服务器结构体。
3. 初始化 Redis 服务器配置。

14.3.1 初始化 Redis 服务器结构体

Redis 服务器结构体 `struct redisServer` 的定义如下：

```
struct redisServer {  
    int runid;  
    int configfile;  
    int hz;  
    int arch_bits;  
};
```

在 `redis.c/initServerConfig` 函数中，我们初始化了 `server` 结构体的成员变量。

```
void initServerConfig(void){  
    //  
    初始化 runid  
    getRandomHexChars(server.runid,REDIS_RUN_ID_SIZE);  
    //  
    初始化 id  
    初始化 configfile  
    server.runid[REDIS_RUN_ID_SIZE] = '\0';  
    //  
    初始化 hz  
    server.configfile = NULL;  
    //  
    初始化 hz  
    server.hz = REDIS_DEFAULT_HZ;  
    //  
    初始化 arch_bits  
    server.arch_bits = (sizeof(long) == 8) ? 64 : 32;  
    //  
}
```

```
server.port = REDIS_SERVERPORT;
// ...
}
```

initServerConfig

- ID
-
-
-
-
- RDB AOF
- LRU
-

initServerConfig

initServerConfig

Lua

initServerConfig

14.3.2 安装Redis

Redis 是一个开源的内存数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希等。Redis 的特点包括高性能、持久化、分布式等。

安装 Redis 的步骤如下：

```
$ redis-server --port 10086
```

Redis 的配置文件 redis.conf 位于 /etc/redis/ 目录下。默认情况下，Redis 使用 6379 端口。

```
$ redis-server redis.conf
```

Redis 的配置文件 redis.conf 包含以下配置项：

```
#
# 设置 Redis 的数据库数量
#
databases 32
#
# 设置 Redis 的持久化策略
#
rdbcompression no
```

Redis 的持久化策略包括 RDB 和 AOF。RDB 是 Redis 的默认持久化策略，它会将 Redis 的数据快照保存到磁盘。

Redis 的配置文件 redis.conf 包含以下配置项：

```
initServerConfig server
```

Redis 的配置文件 redis.conf 包含以下配置项：

```
server
```

Redis 的配置文件 redis.conf 包含以下配置项：

```
server port
```

```
void initServerConfig(void){
    // ...
    //
    6379
    server.port = REDIS_SERVERPORT;
    // ...
}
```

redis.conf 中的 port 10086 与 server.port 10086 一致，而 6379 与 10086 不一致。

redis.conf 中的 server.dbnum 与 server.dbnum 一致。

```
void initServerConfig(void){
    // ...
    //
    16
    server.dbnum = REDIS_DEFAULT_DBNUM;
    // ...
}
```

redis.conf 中的 databases 32 与 server.dbnum 32 一致，而 16 与 32 不一致。

redis.conf 中的 port 与 dbnum 一致。

· redis.conf 中的 server.port 与 server.dbnum 一致。

· redis.conf 中的 server.dbnum 与 server.dbnum 一致。

redis.conf

server 的初始化函数，
——

14.3.3

initServerConfig 的初始化函数，
——

server.clients 的初始化函数，
redisClient

server.db 的初始化函数

server.pubsub_channels 的初始化函数，
server.pubsub_patterns

Lua 的初始化函数 server.lua

server.slowlog 的初始化函数

initServer 的初始化函数，
——

initServerConfig 的初始化函数，
——

serverinitServerConfig
initServer

initServer

·

·Redis"OK"
"ERR"110000
Redis

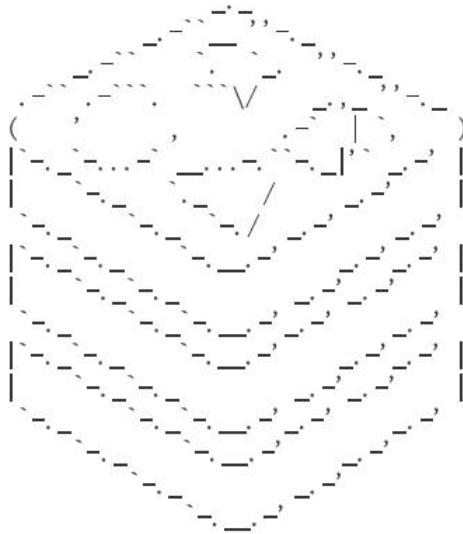
·
Redis

·serverCronserverCron

·AOF
AOF
AOF

·IObioIO

initServerASCIIRedis
Redis



Redis 2.9.11 (b139a2ac/0) 64 bit

Running in stand alone mode

Port: 6379

PID: 5244

<http://redis.io>

[5244] 21 Nov 22:43:49.084 # Server started, Redis version 2.9.11

14.3.4 持久化

Redis server 提供了两种持久化方案 RDB 和 AOF。RDB 是 Redis 的默认持久化方案，它会将 Redis 中的数据快照保存到磁盘上。

AOF 是 Redis 的另一种持久化方案，它会将 Redis 中的每个写操作都记录到磁盘上。

Redis 提供了两种持久化方案 AOF 和 RDB。AOF 是 Redis 的默认持久化方案，它会将 Redis 中的每个写操作都记录到磁盘上。

RDB 是 Redis 的另一种持久化方案，它会将 Redis 中的数据快照保存到磁盘上。

Redis 提供了两种持久化方案 AOF 和 RDB。AOF 是 Redis 的默认持久化方案，它会将 Redis 中的每个写操作都记录到磁盘上。

[5244] 21 Nov 22:43:49.084 * DB loaded from disk: 0.068 seconds

14.3.5 配置

```
[5244] 21 Nov 22:43:49.084 * The server is now ready to accept connections
on port 6379
```

[illegible]

14.4

· 1 2 3 4

· serverCron 100 SIGTERM

· 1 2 3 4 5

□□□□ □□□□□□□□

□15□ □□

□16□ Sentinel

□17□ □□

15 主从复制

Redis 提供了主从复制功能，通过 `SLAVEOF` 命令或 `slaveof` 配置项，主服务器可以复制到其他从服务器。
`replicate` 命令用于在主服务器上配置从服务器，`master` 命令用于在从服务器上配置主服务器。
本节将介绍 Redis 主从复制的配置和原理。

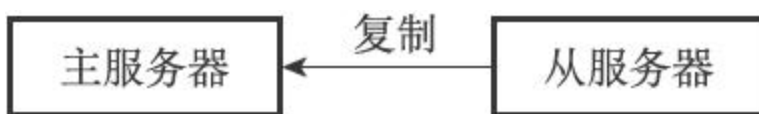


图 15-1 主从复制示意图

本节将介绍 Redis 主从复制的配置和原理。
127.0.0.1:12345 为主服务器，127.0.0.1:12345 为从服务器。

```
127.0.0.1:12345> SLAVEOF 127.0.0.1 6379
OK
```

主服务器 127.0.0.1:12345 向从服务器 127.0.0.1:6379 复制数据。
从服务器 127.0.0.1:6379 向主服务器 127.0.0.1:12345 复制数据。

主从复制的原理是，主服务器将数据复制到从服务器，从服务器再将数据复制到主服务器。
“主从复制”是指主服务器和从服务器之间的复制。

主从复制的原理是，主服务器将数据复制到从服务器，从服务器再将数据复制到主服务器。

127.0.0.1:6379> SET msg "hello world"
OK

redis-cli -h 127.0.0.1 -p 6379 --raw
msg

127.0.0.1:6379> GET msg
"hello world"

redis-cli -h 127.0.0.1 -p 6379 --raw
msg

127.0.0.1:12345> GET msg
"hello world"

redis-cli -h 127.0.0.1 -p 12345 --raw
msg

127.0.0.1:6379> DEL msg
(integer) 1

redis-cli -h 127.0.0.1 -p 6379 --raw
msg

127.0.0.1:6379> EXISTS msg
(integer) 0

redis-cli -h 127.0.0.1 -p 6379 --raw
msg

127.0.0.1:12345> EXISTS msg
(integer) 0

Redis

<http://redis.io/topics/replication>

Redis 2.8

Redis 2.8

SLAVEOF

15.1 主从复制

Redis 主从复制的同步(sync)过程分为两个阶段: command propagate 和 full sync。

- 主从复制的同步(sync)过程分为两个阶段: command propagate 和 full sync。

- 主从复制的同步(sync)过程分为两个阶段: command propagate 和 full sync。

命令传播(command propagate)阶段

主从复制的同步(sync)过程分为两个阶段: command propagate 和 full sync。

15.1.1 命令传播

主从复制的同步(sync)过程分为两个阶段: SLAVEOF 命令传播和 full sync。

主从复制的同步(sync)过程分为两个阶段: SYNC 命令传播和 full sync。

1. 主从复制的同步(sync)过程分为两个阶段: SYNC 命令传播和 full sync。

2. 主从复制的同步(sync)过程分为两个阶段: SYNC 命令传播和 full sync。

3 主服务器执行BGSAVE命令，将内存中的数据集写入到RDB文件中。
4 主服务器执行BGSAVE命令，将内存中的数据集写入到RDB文件中。
5 主服务器执行BGSAVE命令，将内存中的数据集写入到RDB文件中。

4 主服务器执行BGSAVE命令，将内存中的数据集写入到RDB文件中。
5 主服务器执行BGSAVE命令，将内存中的数据集写入到RDB文件中。

图15-2 主服务器发送SYNC命令给从服务器



图15-2 主服务器发送SYNC命令给从服务器

图15-1 主服务器发送SYNC命令给从服务器

图15-1 主服务器发送SYNC命令给从服务器

图15-4

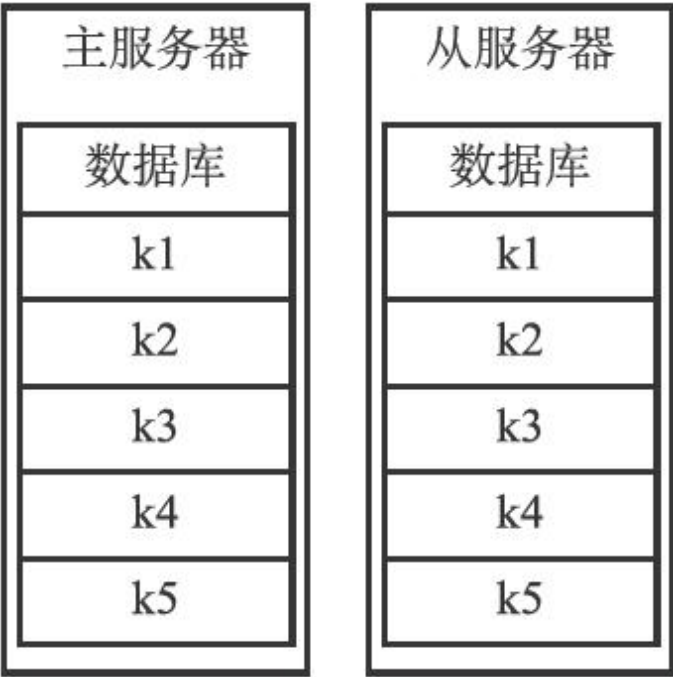


图15-3 数据库同步示意图

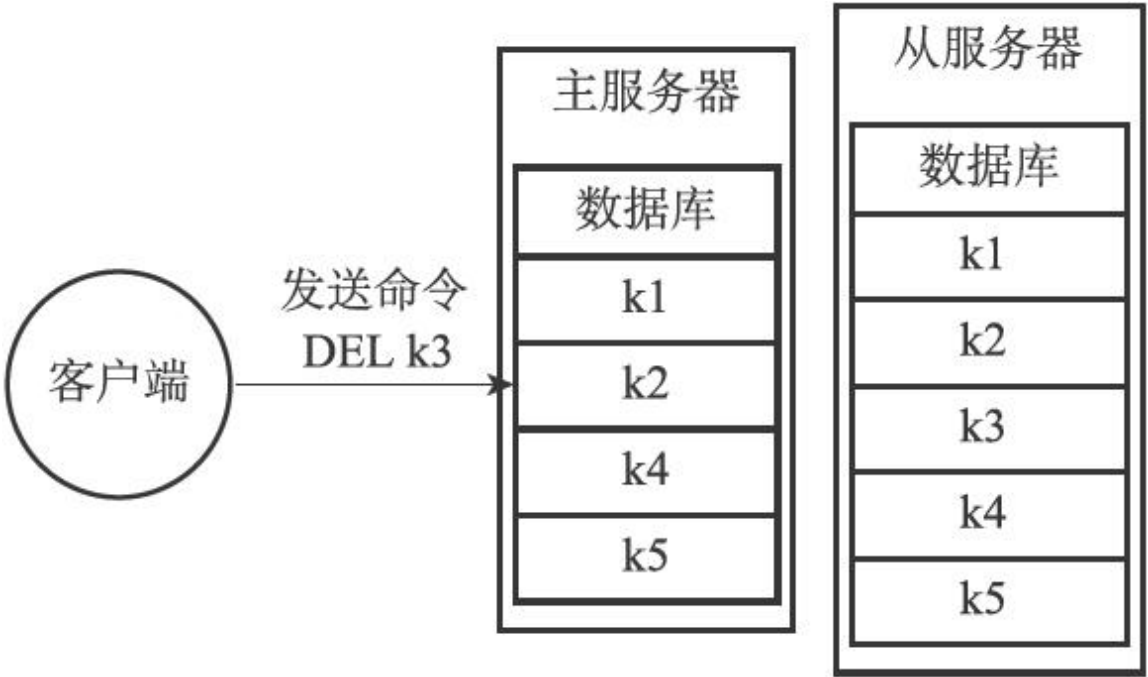


图15-4 数据库同步示意图

数据库主服务器和从服务器都包含数据库，数据库包含数据k1、k2、k4、k5，主服务器还包含数据k3，从服务器不包含数据k3。

主服务器向从服务器发送DEL k3命令，从服务器执行DEL k3命令，从服务器数据库不包含数据k3。

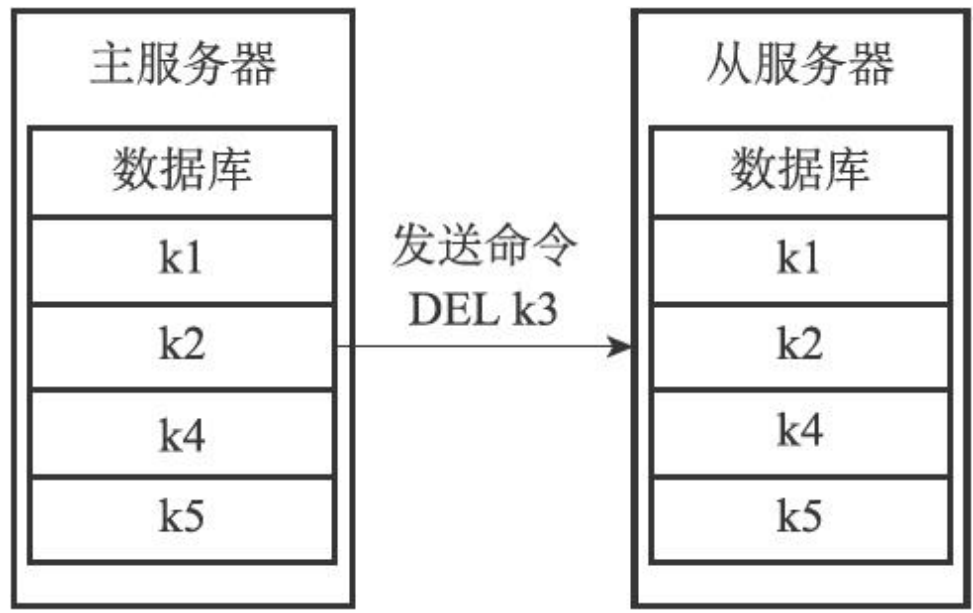


图15-5 数据库同步示意图

15.2 Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库

Redis 数据库 15-2

15-2 Redis 数据库

时间	主服务器	从服务器
T0	主从服务器完成同步	主从服务器完成同步
T1	执行并传播 SET k1 v1	执行主服务器传来的 SET k1 v1
T2	执行并传播 SET k2 v2	执行主服务器传来的 SET k2 v2
...
T10085	执行并传播 SET k10085 v10085	执行主服务器传来的 SET k10085 v10085
T10086	执行并传播 SET k10086 v10086	执行主服务器传来的 SET k10086 v10086
T10087	主从服务器连接断开	主从服务器连接断开
T10088	执行 SET k10087 v10087	断线中，尝试重新连接主服务器
T10089	执行 SET k10088 v10088	断线中，尝试重新连接主服务器
T10090	执行 SET k10089 v10089	断线中，尝试重新连接主服务器
T10091	主从服务器重新连接	主从服务器重新连接
T10092		向主服务器发送 SYNC 命令
T10093	接收到从服务器发来的 SYNC 命令，执行 BGSAVE 命令，创建包含键 k1 至键 k10089 的 RDB 文件，并使用缓冲区记录接下来执行的所有写命令	
T10094	BGSAVE 命令执行完毕，向从服务器发送 RDB 文件	
T10095		接收并载入主服务器发来的 RDB 文件，获得键 k1 至键 k10089
T10096	因为在 BGSAVE 命令执行期间，主服务器没有执行任何写命令，所以跳过发送缓冲区包含的写命令这一步	
T10097	主从服务器再次完成同步	主从服务器再次完成同步

在 T10091 时刻，主从服务器重新连接成功。主服务器向从服务器发送 SYNC 命令，从服务器接收并执行 SYNC 命令，创建包含键 k1 至键 k10089 的 RDB 文件，并使用缓冲区记录接下来执行的所有写命令。

在 T10092 时刻，从服务器向主服务器发送 SYNC 命令，主服务器接收并执行 SYNC 命令，接收并载入主服务器发来的 RDB 文件，获得键 k1 至键 k10089。

·T0T10086

·
k10087k10088k10089

·
k1k10089RDBRDBk1k10086
RDB

Diagram illustrating a 40-bit bus structure. The bus is divided into four 10-bit segments. The first three segments are labeled 'DATA' and the fourth is labeled 'SYNC'.

SYNC

□□□□SYNC□□□□□□□□□□□□□□□□

1 BGSAVE RDB CPU I/O

2 RDB

3 RDB RDB
RDB

SYNC Redis
SYNC

15.3 复制与同步

Redis 2.8 引入了 PSYNC 命令，用于实现主从复制的增量同步。在 Redis 2.8 之前，主从复制只能通过 SYNC 命令进行全量同步。

PSYNC 命令支持两种同步模式：full resynchronization（全量同步）和 partial resynchronization（增量同步）。

- 当主从复制首次建立连接时，从节点会向主节点发送 PSYNC 命令，请求进行全量同步。此时，主节点会将整个数据集复制到从节点。

- 当主节点对数据集进行了修改后，从节点会向主节点发送 PSYNC 命令，请求进行增量同步。此时，主节点只会将修改后的数据复制到从节点。

PSYNC 命令的格式如下：
PSYNC [flags] [start_offset] [end_offset]

图 15-3 展示了 PSYNC 命令的执行过程。

时间	主服务器	从服务器
T0	主从服务器完成同步	主从服务器完成同步
T1	执行并传播 SET k1 v1	执行主服务器传来的 SET k1 v1
T2	执行并传播 SET k2 v2	执行主服务器传来的 SET k2 v2
...
T10085	执行并传播 SET k10085 v10085	执行主服务器传来的 SET k10085 v10085
T10086	执行并传播 SET k10086 v10086	执行主服务器传来的 SET k10086 v10086
T10087	主从服务器连接断开	主从服务器连接断开
T10088	执行 SET k10087 v10087	断线中，尝试重新连接主服务器
T10089	执行 SET k10088 v10088	断线中，尝试重新连接主服务器
T10090	执行 SET k10089 v10089	断线中，尝试重新连接主服务器
T10091	主从服务器重新连接	主从服务器重新连接
T10092		向主服务器发送 PSYNC 命令
T10093	向从服务器返回 +CONTINUE 回复，表示执行部分重同步	
T10094		接收 +CONTINUE 回复，准备执行部分重同步
T10095	向从服务器发送 SET k10087 v10087、SET k10088 v10088、SET k10089 v10089 三个命令	
T10096		接收并执行主服务器传来的三个 SET 命令
T10097	主从服务器再次完成同步	主从服务器再次完成同步

主服务器 SYNC 从服务器 PSYNC 从服务器向主服务器 SYNC 从服务器
 PSYNC 从服务器向主服务器 SYNC 从服务器
 从服务器 SYNC 从服务器 RDB 从服务器
 从服务器

图 15-6 主从服务器同步过程

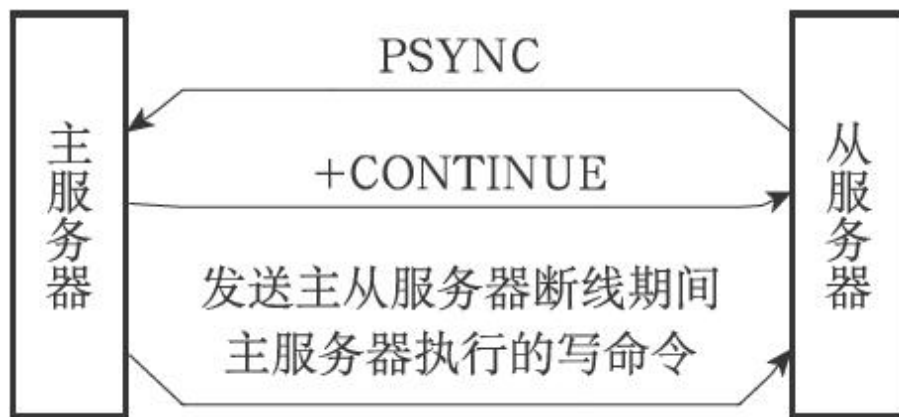


图15-6 主从服务器断线期间的通信

15.4 数据同步

数据同步PSYNC命令用于在从库和主库之间同步数据。该命令的格式如下：

PSYNC [master_host] [master_port] [master_run_id]

- [master_host] 主库的主机名
- [master_port] 主库的端口号
- [master_run_id] 主库的run ID

该命令的返回值如下：

15.4.1 数据同步

数据同步命令的格式如下：

- 主库的run ID为N，从库的run ID为N
- 主库的run ID为N，从库的run ID为N

图15-7 数据同步命令的格式

10086



图15-7 数据同步示意图

数据同步时，主服务器将数据复制到从服务器，偏移量为33字节。
 $10086 + 33 = 10119$ ，即从服务器的偏移量为10119。
 10119 - 10086 = 23，即数据同步的字节数为23。

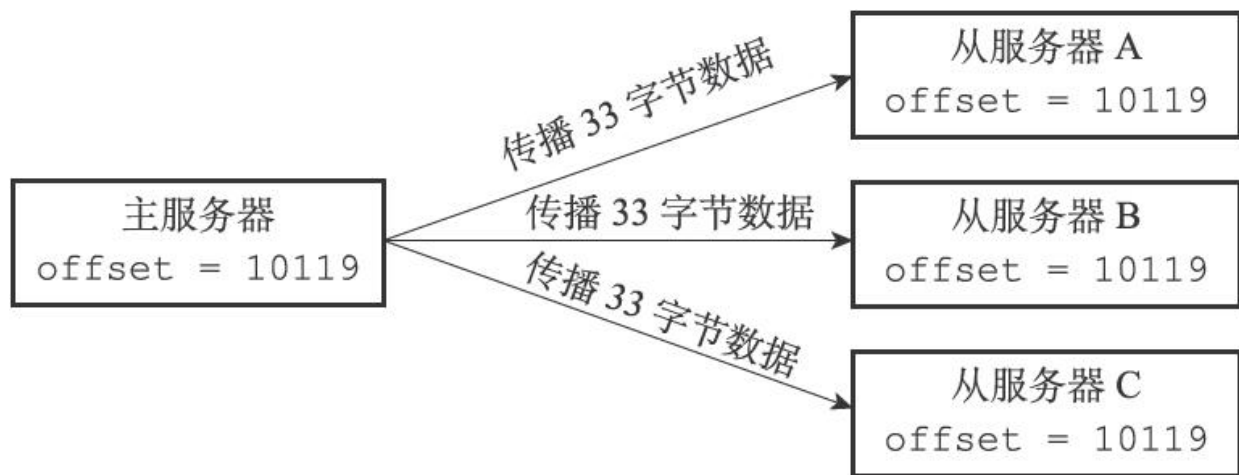


图15-8 数据同步示意图

数据同步时，主服务器将数据复制到从服务器，偏移量为33字节。

· 数据块大小 10086 字节

· 数据块大小 10086 字节

图 15-7 展示了主服务器与从服务器 A、B、C 之间的数据同步过程。主服务器（offset = 10119）向从服务器 A（offset = 10086）发送 33 字节数据。从服务器 A 向从服务器 B（offset = 10119）和从服务器 C（offset = 10119）发送 33 字节数据。从服务器 B 向从服务器 C 发送 33 字节数据。图 15-9 展示了从服务器 A 向主服务器发送数据的过程。

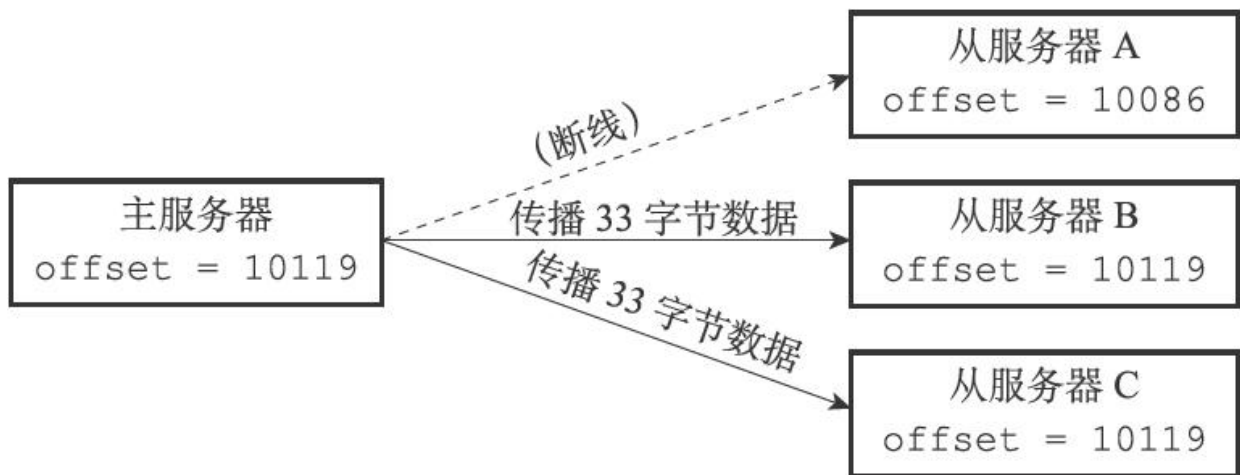


图 15-9 从服务器 A 向主服务器发送数据

从服务器 A 向主服务器发送数据的过程如下：从服务器 A 向主服务器发送 33 字节数据。主服务器向从服务器 A 发送 10086 字节数据。从服务器 A 向主服务器发送 33 字节数据。从服务器 A 向主服务器发送 33 字节数据。

15.4.2 队列的实现

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

队列的实现

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

`['h' 'e' 'l']`

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

`['e' 'l' 'l']`

队列的实现可以采用基于数组的 `fixed-size` 队列或基于 `FIFO` 的链表实现。这里我们采用基于数组的 `fixed-size` 队列实现，其容量为 `1MB`。

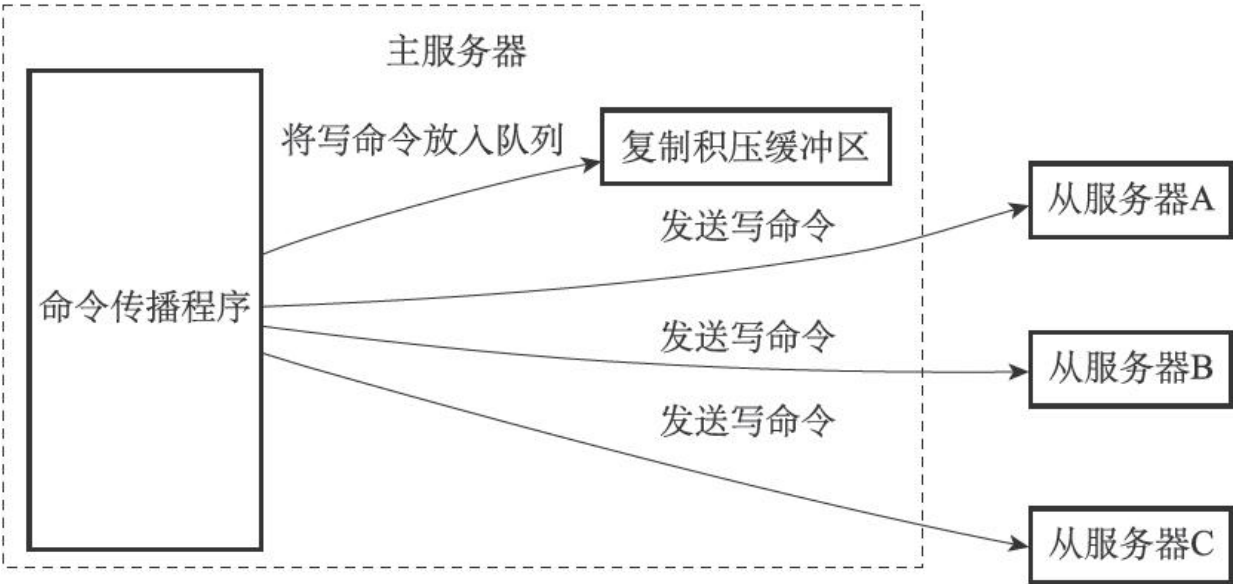
`['l' 'l' 'o']`

[illegible]

☐ ☐

☐ ☐ ☐ ☐ 15-10 ☐ ☐ ☐

□□□□15-10□□□

[illegible]

15-4

15-4

□15-4 □□□□□□□□

偏移量	...	10087	10088	10089	10090	10091	10092	10093	10094	10095	10096	10097	...
字节值	...	'*'	3	'\r'	'\n'	'\$'	3	'\r'	'\n'	'S'	'E'	'T'	...

offset PSYNC offset
offset

offset offset+1
offset

offset
offset

15-9

A PSYNC
10086

PSYNC 10086
10086
+CONTINUE

10086 10087
10119

33 15-11

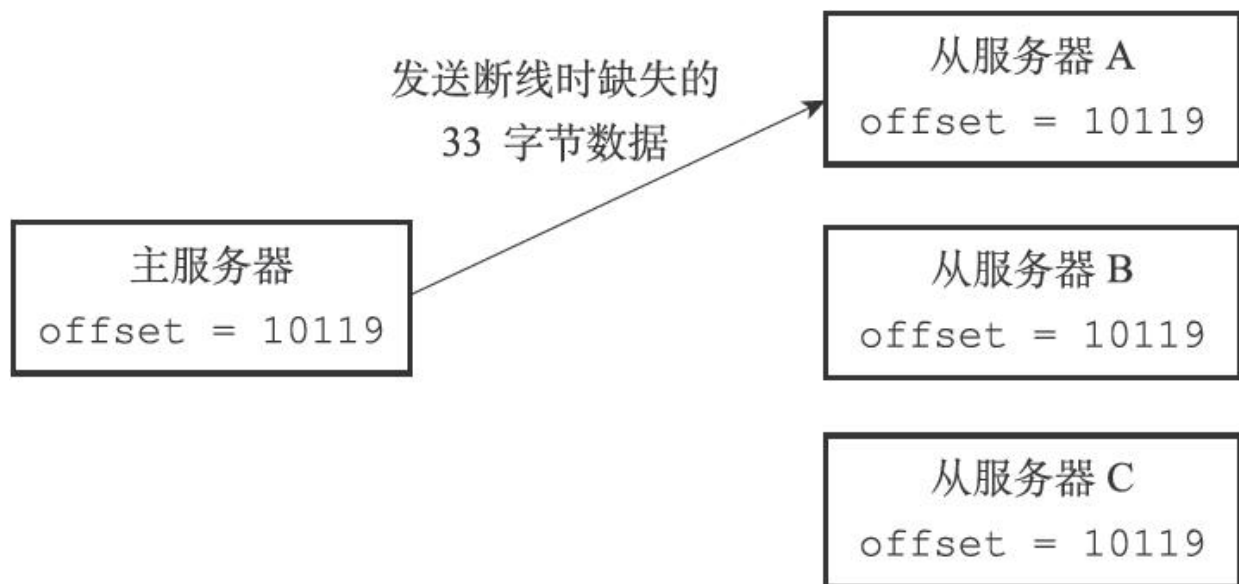


图15-11 数据复制过程示意图

Redis 数据复制过程

Redis 数据复制过程分为两个阶段：主服务器向从服务器发送全量数据（Full Sync）和增量数据（Partial Sync）。

在 Full Sync 阶段，主服务器会发送一个 PSYNC 命令，从服务器会回复一个 PSYNC 命令。主服务器会根据从服务器的回复，计算出需要复制的数据量，并发送相应的数据。

在 Partial Sync 阶段，主服务器会发送一个 PSYNC 命令，从服务器会回复一个 PSYNC 命令。主服务器会根据从服务器的回复，计算出需要复制的数据量，并发送相应的数据。

Redis 数据复制过程的关键参数包括：

- `second*write_size_per_second`：每秒写入的数据量。
- `second`：每秒写入的数据量。
- `write_size_per_second`：每秒写入的数据量。

```
    redis.conf 1 MB redis.conf 5 redis.conf
redis.conf 5MB
```

```
    redis.conf
2*second*write_size_per_second redis.conf
redis.conf
```

```
    redis.conf repl-backlog-size redis.conf
redis.conf
```

15.4.3 Redis ID

Redis ID run ID

Redis ID

ID 40

53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3

ID ID

ID

·IDID
IDID

·IDIDIDID

ID
53b9b28df8042fdc9ab5e3fcbbabff1d5dce2b3
IDID
53b9b28df8042fdc9ab5e3fcbbabff1d5dce2b3
ID

15.5 PSYNC

PSYNCは、レプリケーション中に、主データベースから、従データベースに、最新のレプリケーションIDと、そのIDに属するレプリケーションデータを転送するためのコマンドである。

PSYNCの構文は、以下の通りである。

・`PSYNC SLAVEOF no one` は、主データベースから、従データベースに、最新のレプリケーションIDと、そのIDに属するレプリケーションデータを転送するためのコマンドである。

・`PSYNC <runid> <offset>` は、主データベースから、従データベースに、最新のレプリケーションIDと、そのIDに属するレプリケーションデータを転送するためのコマンドである。

PSYNCの構文は、以下の通りである。

・`PSYNC +FULLRESYNC <runid> <offset>` は、主データベースから、従データベースに、最新のレプリケーションIDと、そのIDに属するレプリケーションデータを転送するためのコマンドである。

redis-cli -h 127.0.0.1 -p 6379
127.0.0.1:6379>redis-cli -h 127.0.0.1 -p 12345

redis-cli -h 127.0.0.1 -p 6379
redis-cli -h 127.0.0.1 -p 12345>PSYNC ? -1

redis-cli -h 127.0.0.1 -p 12345>BGSAVE
+FULLRESYNC
53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3 10086
53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3 ID
10086

redis-cli -h 127.0.0.1 -p 12345>20000
redis-cli -h 127.0.0.1 -p 12345>

redis-cli -h 127.0.0.1 -p 12345>PSYNC
53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3 20000
redis-cli

redis-cli -h 127.0.0.1 -p 12345>PSYNC
ID53b9b28df8042fdc9ab5e3fcbbbabff1d5dce2b3
ID ID ID 20000
redis-cli 20000

□□□□ID□□□□□□□□□□□□□□□□□□□□+CONTINUE□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□20000□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□

15.6 主从复制

主从复制使用 `SLAVEOF` 命令来配置从服务器。

```
SLAVEOF <master_ip> <master_port>
```

例如，将主服务器配置为 `127.0.0.1:12345`。

```
SLAVEOF 127.0.0.1 6379
```

Redis 2.8 及以上版本支持主从复制。

15.6.1 配置主从复制

在从服务器上配置主从复制。

```
127.0.0.1:12345> SLAVEOF 127.0.0.1 6379
OK
```

配置主从复制时，需要指定主服务器的 IP 地址和端口号。

配置主从复制时，需要指定主服务器的 `masterhost` 和 `masterport`。

```
struct redisServer {
    // ...
    //
    masterhost
}
```

```

    char *masterhost;
    //
    int masterport;
    // ...
};

```

图15-13 使用SLAVEOF命令的Redis主从复制配置

redisServer
...
masterhost "127.0.0.1"
masterport 6379
...

图15-13 Redis主从复制配置

SLAVEOF命令的语法如下：
 SLAVEOF masterhost masterport
 其中，masterhost为主服务器的IP地址，masterport为主服务器的端口号。
 执行SLAVEOF命令后，Redis主服务器会返回OK，表示复制配置成功。

15.6.2 配置Redis主从复制

配置Redis主从复制的步骤如下：
 1. 配置主服务器的配置文件，添加以下配置：
 SLAVEOF 127.0.0.1 6379
 2. 配置从服务器的配置文件，添加以下配置：
 SLAVEOF 127.0.0.1 6379

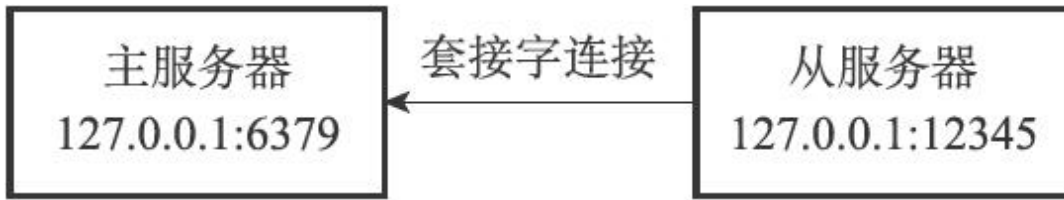


图15-14 套接字连接

主服务器通过调用connect函数与从服务器建立连接。连接成功后，主服务器可以向从服务器发送RDB命令，将主服务器的数据快照传输到从服务器。

从服务器通过调用accept函数接收主服务器的连接。接收成功后，从服务器可以调用server函数启动服务器，并调用client函数接收主服务器的命令。

图15-15



图15-15 命令请求和回复

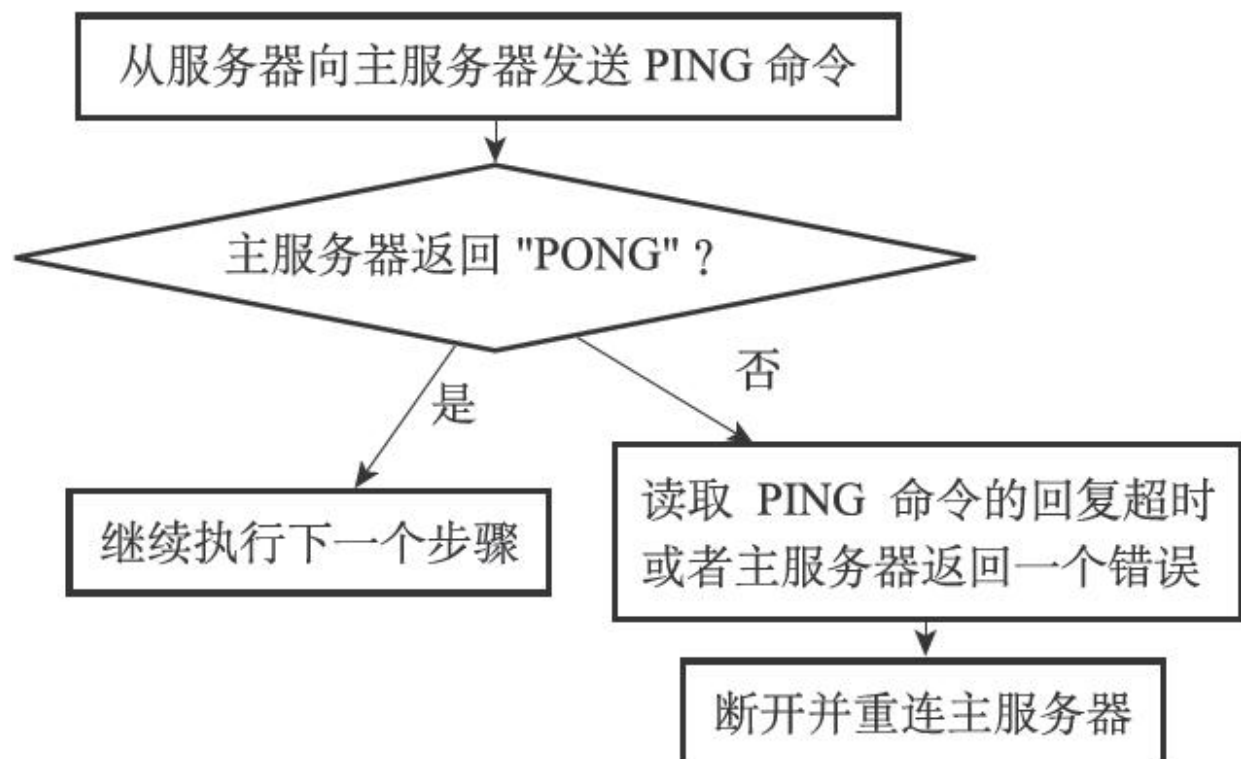
主服务器通过调用send函数向从服务器发送命令请求。从服务器通过调用recv函数接收主服务器的命令请求。从服务器接收到的命令请求格式如下：

15.6.3 3次PING

❑ BUSY Redis is busy running a script. You can only call SCRIPT KILL or SHUTDOWN NOSAVE.❑❑

```
·[0][0][0][0][0]"PONG"[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
```

15-17 PING



15-17 PING

15.6.4 4000000

```
0000000000000000 "PONG" 000000000000000000000000
```

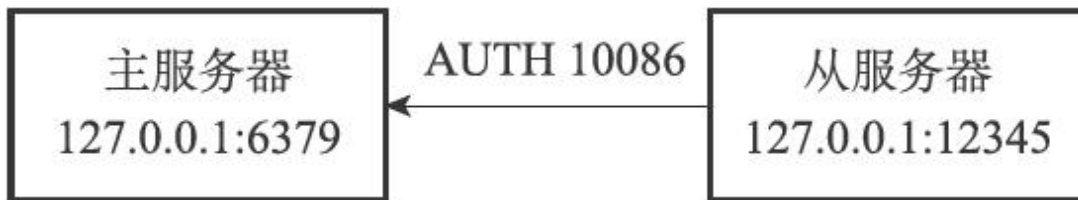
```
·masterauth
```

```
· masterauth
```

[illegible]

masterauth10086

AUTH 1008615-18



□15-18 □□□□□□□□□□

[illegible]

```
·requirepassmasterauth

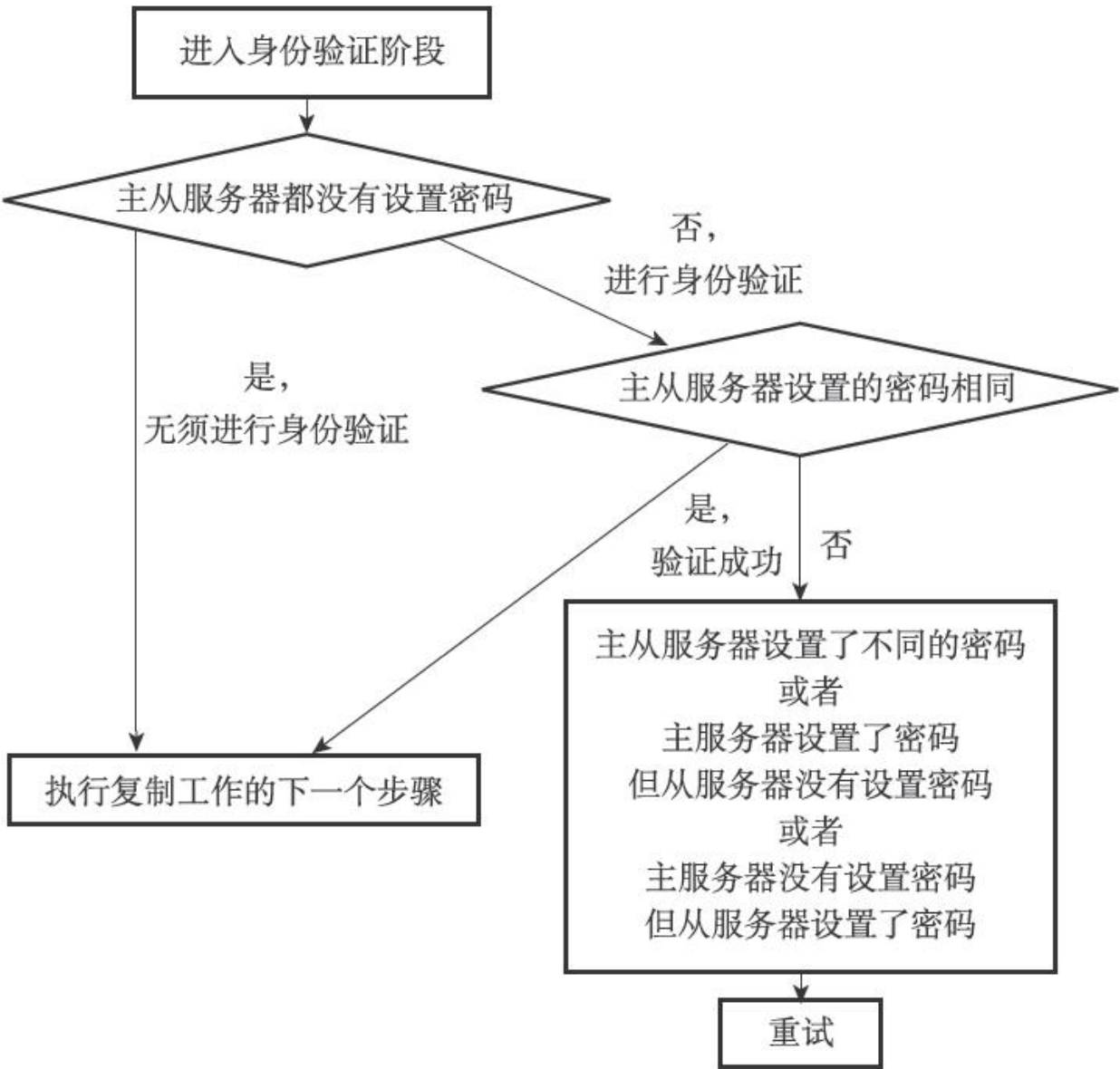
```

```
·AUTHrequirepass
invalid password
```

```
·requirepassmasterauth
NOAUTHrequirepass
```

masterauthno password is set

15-19

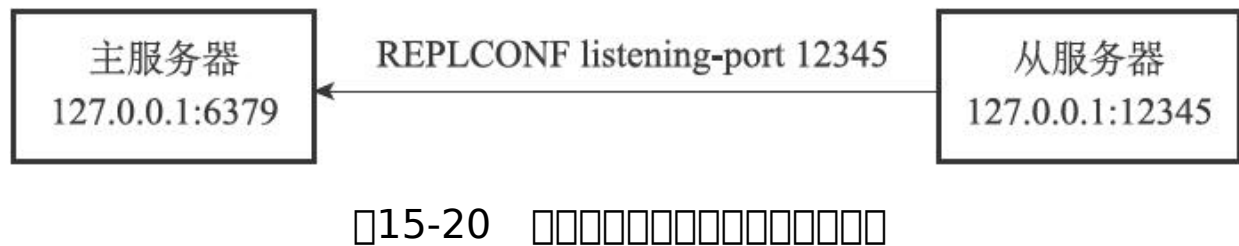


15-19

15.6.5 配置 Redis 主从复制

在 Redis 主服务器上配置 `REPLCONF listening-port <port-number>` 配置项，用于指定主服务器监听从服务器的连接。

在 Redis 从服务器上配置 `REPLCONF listening-port 12345` 配置项，用于指定从服务器监听的连接端口。



在 Redis 从服务器的配置文件中，添加 `slave_listening_port` 配置项，用于指定从服务器监听的连接端口。

```
typedef struct redisClient {  
    // ...  
    //  
    int slave_listening_port;  
    // ...  
} redisClient;
```

图 15-21 配置 Redis 从服务器的 `slave_listening_port` 配置项

redisClient
...
slave_listening_port 12345
...

15-21 复制客户端结构体

slave_listening_port 在 INFO replication 命令的输出中

在 INFO replication 命令的输出中 slave0 的 port 就是 slave_listening_port

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:1
slave0:ip=127.0.0.1,port=12345,status=online,offset=1289,lag=1
master_repl_offset:1289
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:1288
```

15.6.6 复制

复制过程分为 PSYNC 和 FULLRESYNC 两个阶段

1. 主服务器向从服务器发送命令请求/返回命令回复
 2. 从服务器向主服务器发送命令请求/返回命令回复

3. 主服务器向从服务器发送命令请求/返回命令回复
 4. 从服务器向主服务器发送命令请求/返回命令回复

5. 主服务器向从服务器发送命令请求/返回命令回复
 6. 从服务器向主服务器发送命令请求/返回命令回复

7. 主服务器向从服务器发送命令请求/返回命令回复
 8. 从服务器向主服务器发送命令请求/返回命令回复

9. 主服务器向从服务器发送命令请求/返回命令回复
 10. 从服务器向主服务器发送命令请求/返回命令回复

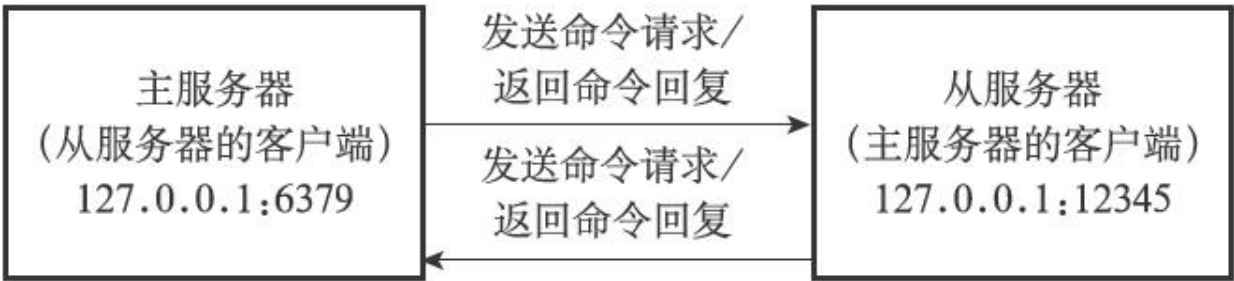


图15-22 主服务器与从服务器的通信

15.6.7 主服务器与从服务器的通信

1. 主服务器向从服务器发送命令请求/返回命令回复
 2. 从服务器向主服务器发送命令请求/返回命令回复

Redis 2.8

15.7 配置

配置项名称

REPLCONF ACK <replication_offset>

replication_offset 配置项

REPLCONF ACK 配置项

- 配置项名称

- 配置项名称 min-slaves

- 配置项名称

配置项名称

15.7.1 配置项名称

配置项名称 REPLCONF ACK 配置项名称

配置项名称 REPLCONF ACK 配置项名称

配置项名称

INFO replicationlag
REPLCONF ACK

```
127.0.0.1:6379> INFO replication
# Replication
role:master
connected_slaves:2
slave0:ip=127.0.0.1,port=12345,state=online,offset=211,lag=0 #
REPLCONF ACK
slave1:ip=127.0.0.1,port=56789,state=online,offset=197,lag=15 # 15
REPLCONF ACK
master_repl_offset:211
repl_backlog_active:1
repl_backlog_size:1048576
repl_backlog_first_byte_offset:2
repl_backlog_histlen:210
```

lag011

15.7.2 min-slaves

Redismin-slaves-to-writemin-slaves-max-lag

```
min-slaves-to-write 3
min-slaves-max-lag 10
```

INFO replication lag

15.7.3 五五五五五

[illegible]

20015-23



□15-23 □□□□□□□□□□

```

SET key value33
233SET key value
233
20015-24

```

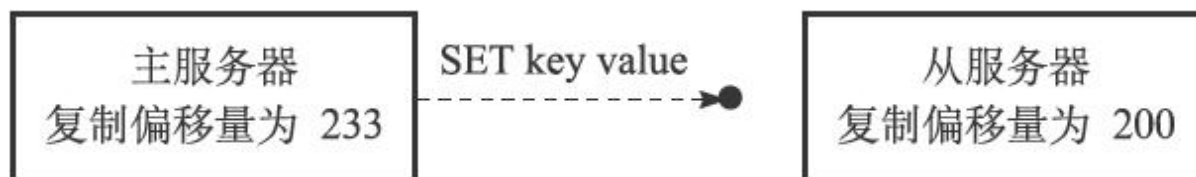


图15-24 主从服务器复制

主服务器向从服务器发送REPLCONF ACK，从服务器收到后，将复制偏移量从200更新为233。主服务器向从服务器发送SET key value，从服务器收到后，将SET key value写入内存。图15-25展示了主从服务器复制的示意图。

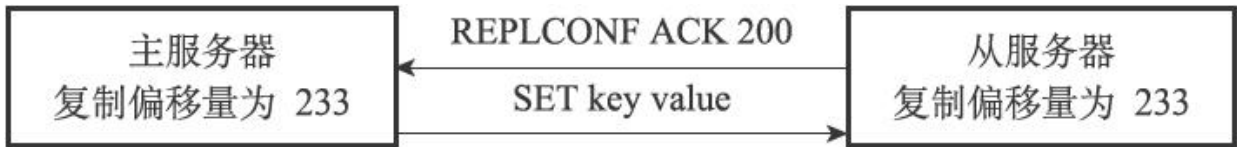


图15-25 主从服务器复制示意图

主服务器向从服务器发送REPLCONF ACK，从服务器收到后，将复制偏移量从200更新为233。主服务器向从服务器发送SET key value，从服务器收到后，将SET key value写入内存。图15-25展示了主从服务器复制的示意图。

Redis 2.8主从服务器复制

REPLCONF ACK，从服务器收到后，将复制偏移量从200更新为233。主服务器向从服务器发送SET key value，从服务器收到后，将SET key value写入内存。图15-25展示了主从服务器复制的示意图。

15.8 Redis

- Redis 2.8에서 Redis 2.8로 업그레이드하는 방법
- Redis 2.8에서 Redis 2.8로 업그레이드하는 방법
- Redis 2.8에서 Redis 2.8로 업그레이드하는 방법
- Redis 2.8에서 Redis 2.8로 업그레이드하는 방법

第16章 Sentinel

Sentinel是Redis的高可用性（high availability）解决方案。Sentinel系统由多个Sentinel实例组成，它们通过互相通信来监控Redis主从节点的状态。当主节点发生故障时，Sentinel系统会自动将流量切换到从节点，从而实现故障转移（failover）。此外，Sentinel还可以配置为自动创建从节点，以在主节点故障时快速恢复服务。

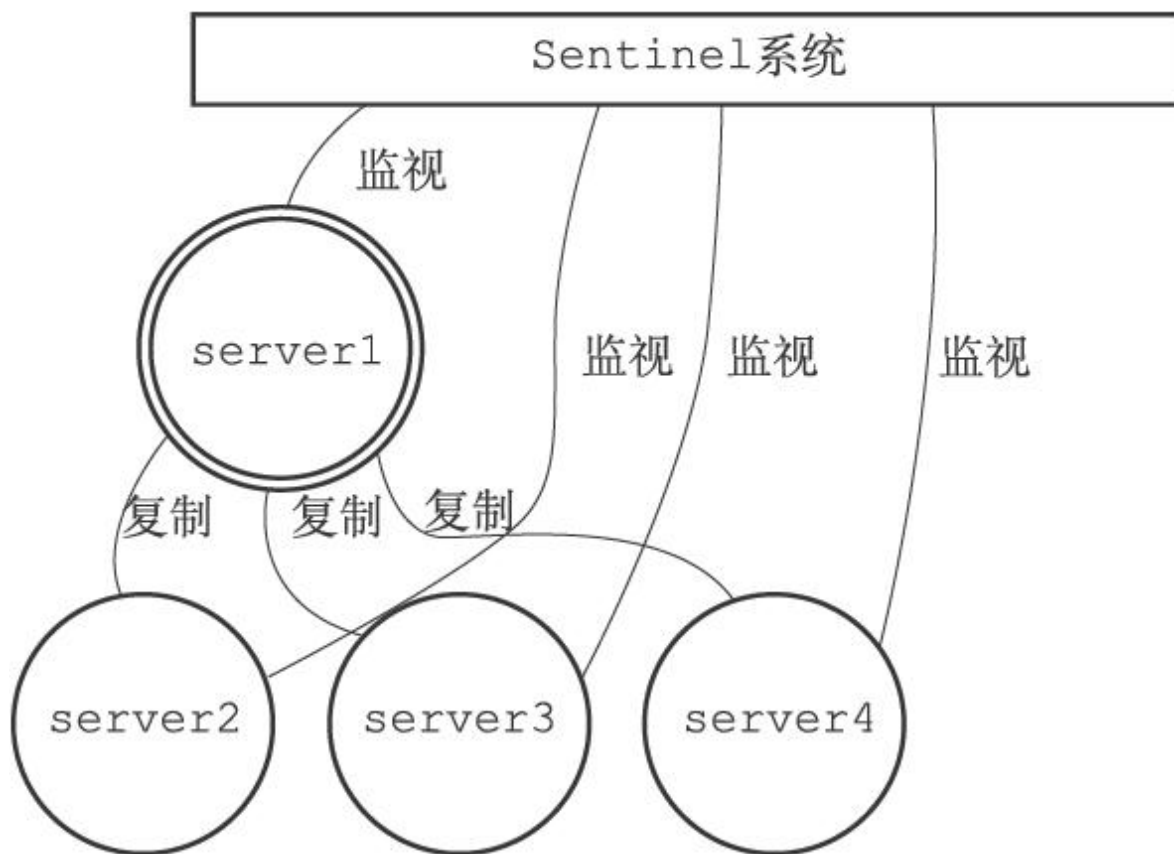


图16-1 Sentinel系统

图16-1展示了Sentinel系统的基本架构。

```
·server1
```

```
·server2server3server4
```

```
·server2server3server4server1
```

Sentinel□□□□□□□□□□

```
server1server2server3
server4Sentinelserver116-
2
```

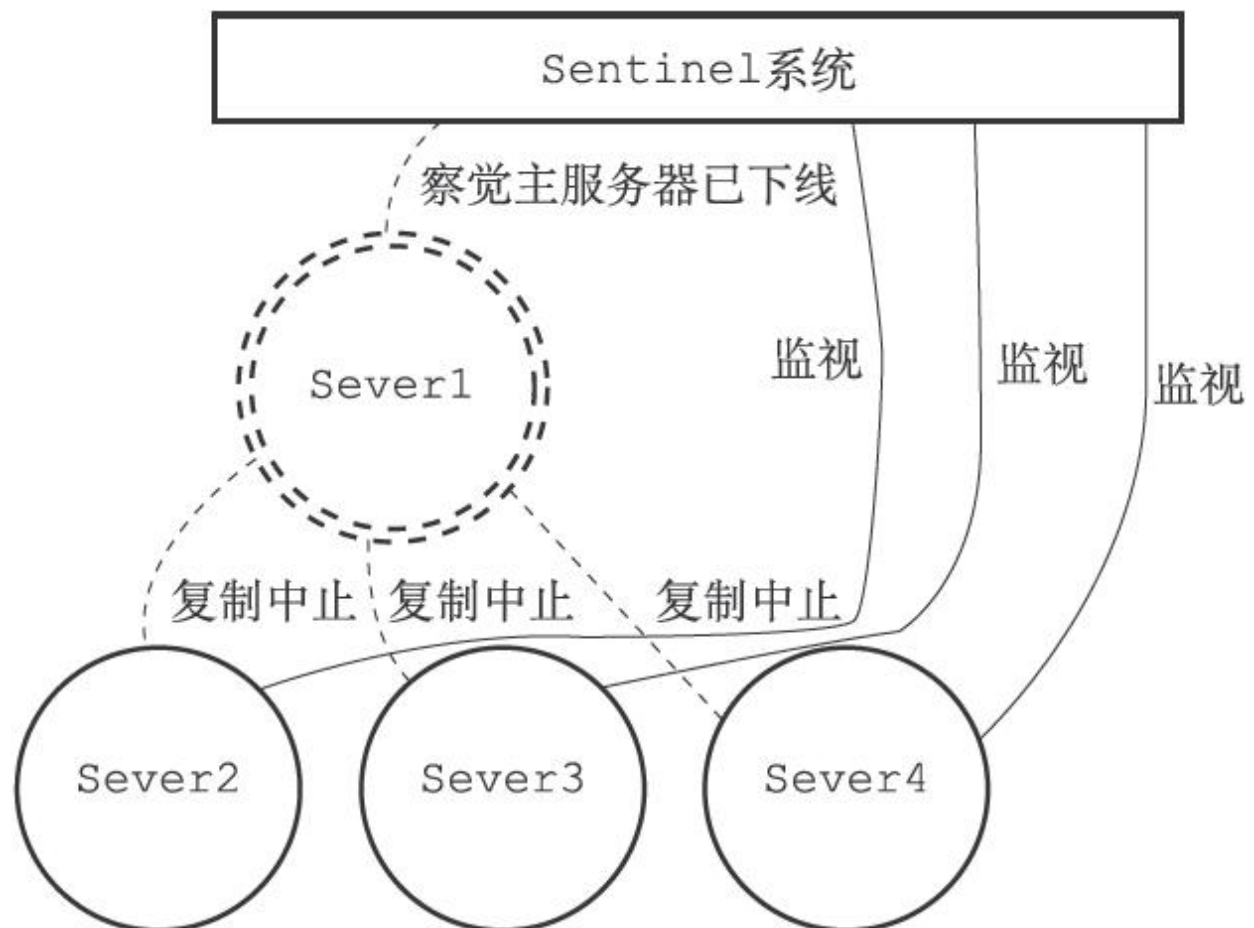


图16-2 哨兵配置

在server1上配置哨兵，配置文件中添加以下配置，并重启server1，使配置生效。

·在server1的配置文件sentinel.conf中添加以下配置，并重启server1，使配置生效。

·在server1的配置文件sentinel.conf中添加以下配置，并重启server1，使配置生效。

·在server1的配置文件sentinel.conf中添加以下配置，并重启server1，使配置生效。

图16-3 哨兵配置server2、server3、server4，使server2成为主节点。

在server1上配置哨兵，配置文件中添加以下配置，并重启server1，使配置生效。

16-4 哨兵

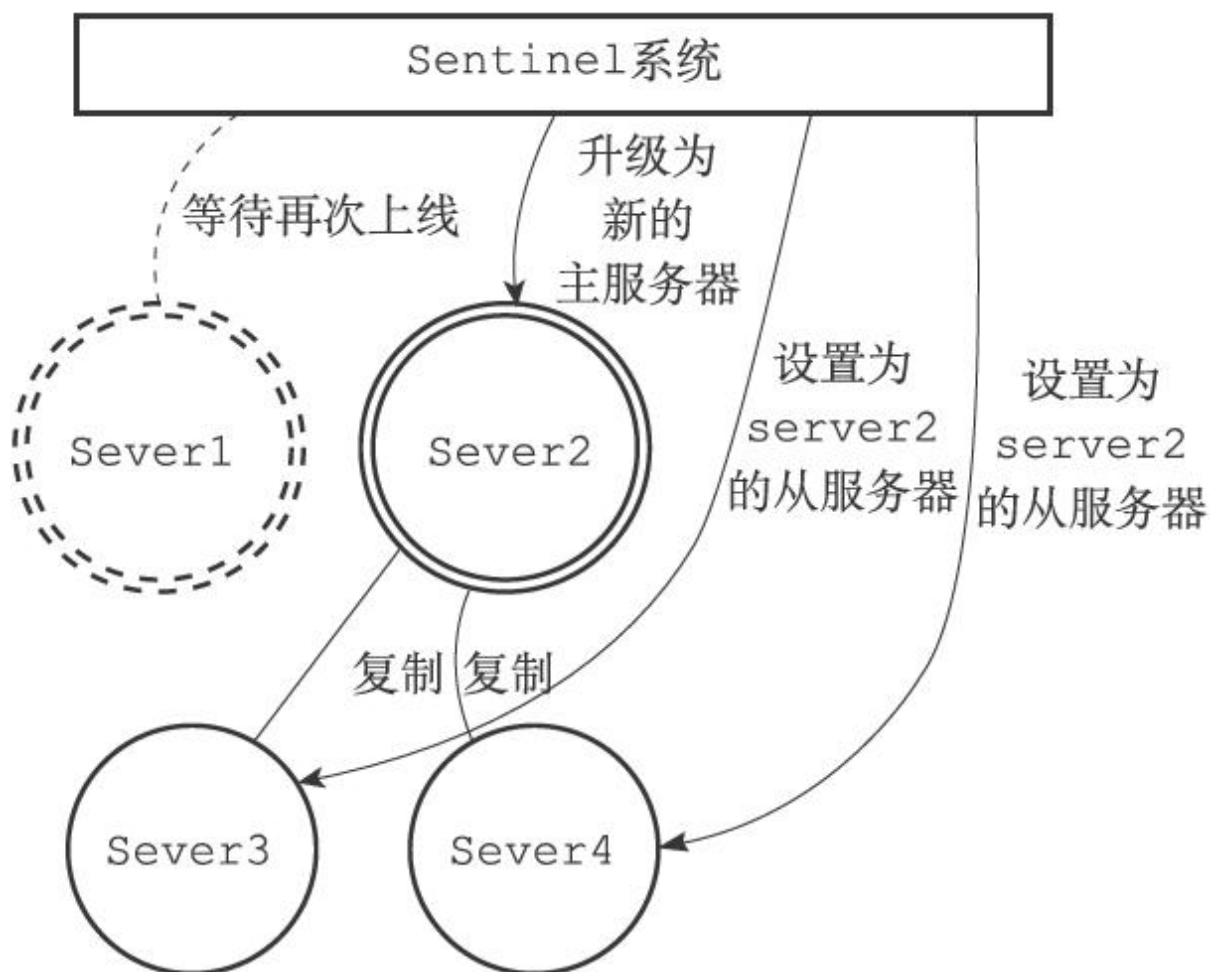


图16-3 哨兵

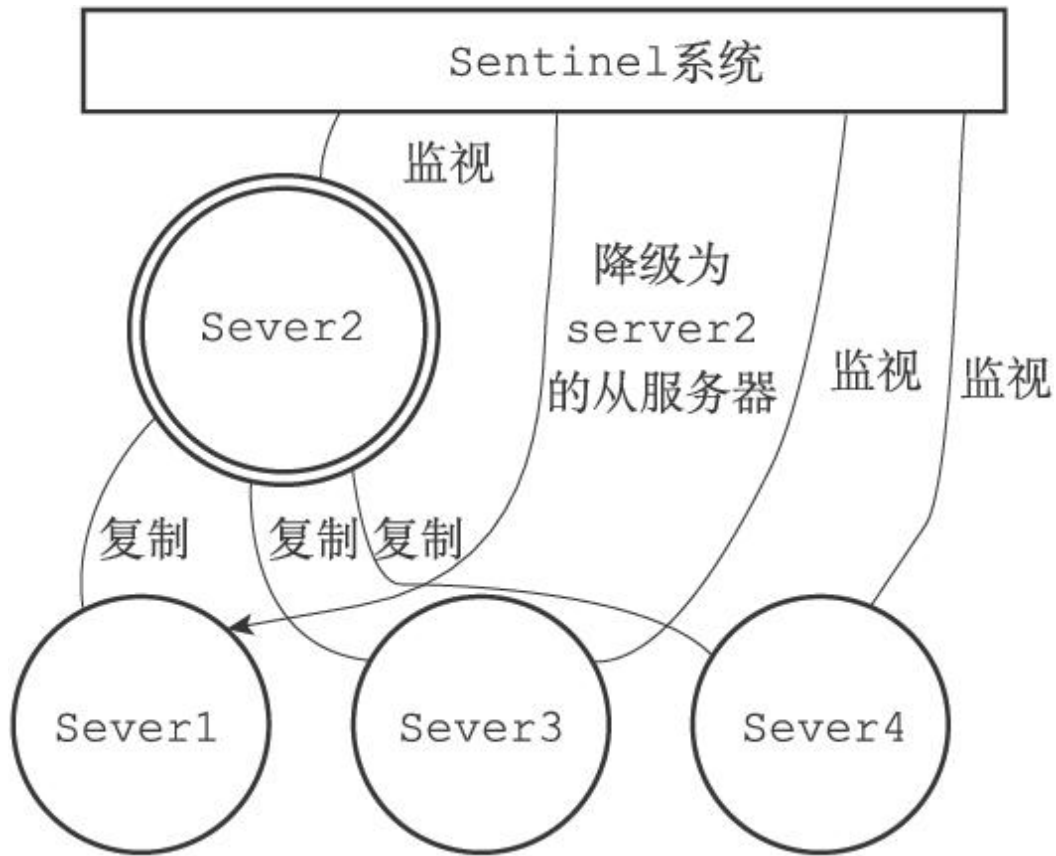


图16-4 Redis哨兵系统

Redis哨兵系统由多个Sentinel实例组成，每个Sentinel实例都监控着Redis主从实例。当主实例发生故障时，Sentinel实例会检测到故障，并自动将主实例的角色切换到从实例上，从而实现高可用性。

Redis哨兵系统的工作原理如下：每个Sentinel实例都会定期向被监控的Redis主从实例发送心跳。如果主实例在规定的时间内没有响应，Sentinel实例就会判定主实例发生故障。此时，Sentinel实例会启动故障转移流程，将主实例的角色切换到从实例上。

Redis哨兵系统还支持配置多个Sentinel实例，以提高系统的可靠性和可用性。

16.1 Sentinel

Redis Sentinel

```
$ redis-sentinel /path/to/your/sentinel.conf
```

Redis

```
$ redis-server /path/to/your/sentinel.conf --sentinel
```

Redis Sentinel

Redis Sentinel

1 Redis

2 Redis Sentinel

3 Redis Sentinel

4 Redis Sentinel

5 Redis

Redis Sentinel

16.1.1 哨兵

哨兵 Sentinel 是 Redis 的高可用解决方案。哨兵 Redis 是 Redis 14 版本引入的。

哨兵 Sentinel 是 Redis 的高可用解决方案。哨兵 Sentinel 是 Redis 的高可用解决方案。

哨兵 Sentinel 是 Redis 的高可用解决方案。哨兵 Sentinel 是 Redis 的高可用解决方案。

图 16-1 哨兵 Redis 的高可用解决方案

表 16-1 Sentinel 与 Redis 的命令支持情况

功 能	使用情况
数据库和键值对方面的命令，比如 <i>SET</i> 、 <i>DEL</i> 、 <i>FLUSHDB</i>	不使用
事务命令，比如 <i>MULTI</i> 和 <i>WATCH</i>	不使用
脚本命令，比如 <i>EVAL</i>	不使用
RDB 持久化命令，比如 <i>SAVE</i> 和 <i>BGSAVE</i>	不使用
AOF 持久化命令，比如 <i>BGREWRITEAOF</i>	不使用
复制命令，比如 <i>SLAVEOF</i>	Sentinel 内部可以使用，但客户端不可以使用
发布与订阅命令，比如 <i>PUBLISH</i> 和 <i>SUBSCRIBE</i>	<i>SUBSCRIBE</i> 、 <i>PSUBSCRIBE</i> 、 <i>UNSUBSCRIBE</i> 、 <i>PUNSUBSCRIBE</i> 四个命令在 Sentinel 内部和客户端都可以使用，但 <i>PUBLISH</i> 命令只能在 Sentinel 内部使用
文件事件处理器（负责发送命令请求、处理命令回复）	Sentinel 内部使用，但关联的文件事件处理器和普通 Redis 服务器不同
时间事件处理器（负责执行 <i>serverCron</i> 函数）	Sentinel 内部使用，时间事件的处理器仍然是 <i>serverCron</i> 函数， <i>serverCron</i> 函数会调用 <i>sentinel.c/sentinelTimer</i> 函数，后者包含了 Sentinel 要执行的所有操作

16.1.2 Sentinel

Sentinel Redis Sentinel
Redis redis.h/REDIS_SERVERPORT

```
#define REDIS_SERVERPORT 6379
```

Sentinel sentinel.c/REDIS_SENTINEL_PORT

```
#define REDIS_SENTINEL_PORT 26379
```

Redis redis.c/redisCommandTable

```
struct redisCommand redisCommandTable[] = {  
    {"get",getCommand,2,"r",0,NULL,1,1,1,0,0},  
    {"set",setCommand,-3,"wm",0,noPreloadGetKeys,1,1,1,0,0},  
    {"setnx",setnxCommand,3,"wm",0,noPreloadGetKeys,1,1,1,0,0},  
    // ...  
    {"script",scriptCommand,-2,"ras",0,NULL,0,0,0,0,0},  
    {"time",timeCommand,1,"rR",0,NULL,0,0,0,0,0},  
    {"bitop",bitopCommand,-4,"wm",0,NULL,2,-1,1,0,0},  
    {"bitcount",bitcountCommand,-2,"r",0,NULL,1,1,1,0,0}  
}
```

哨兵 Sentinel 的 sentinel.c/sentinelcmds 文件中定义了哨兵支持的命令。INFO 命令在 Sentinel 的 sentinel.c/sentinelInfoCommand 文件中定义。Redis 的 redis.c/infoCommand 文件中

```
struct redisCommand sentinelcmds[] = {
    {"ping", pingCommand, 1, "", 0, NULL, 0, 0, 0, 0},
    {"sentinel", sentinelCommand, -2, "", 0, NULL, 0, 0, 0, 0},
    {"subscribe", subscribeCommand, -2, "", 0, NULL, 0, 0, 0, 0},
    {"unsubscribe", unsubscribeCommand, -1, "", 0, NULL, 0, 0, 0, 0},
    {"psubscribe", psubscribeCommand, -2, "", 0, NULL, 0, 0, 0, 0},
    {"punsubscribe", punsubscribeCommand, -1, "", 0, NULL, 0, 0, 0, 0},
    {"info", sentinelInfoCommand, -1, "", 0, NULL, 0, 0, 0, 0}
};
```

sentinelcmds 文件中定义的哨兵支持的命令包括 Sentinel、Redis 的 SET、DBSIZE、EVAL、PING、SENTINEL、INFO、SUBSCRIBE、UNSUBSCRIBE、PSUBSCRIBE、PUNSUBSCRIBE 等命令。哨兵 Sentinel 的

16.1.3 哨兵 Sentinel

哨兵 Sentinel 的 sentinel.c/sentinelState 文件中定义了哨兵的状态。哨兵 Sentinel 的 redis.h/redisServer 文件中

```
struct sentinelState {
    //
    uint64_t current_epoch;
    //
```

```

struct sentinel
{
    //
    struct redisInstance *
    //
    struct redisInstance *sentinelRedisInstance
    struct dict *
        dict *masters;
    //
    struct tilt
    int tilt;
    //
    struct running_scripts
    int running_scripts;
    //
    struct tilt
    struct tilt
    mstime_t tilt_start_time;
    //
    struct previous_time
    mstime_t previous_time;
    //
    struct FIFO
    struct scripts_queue
    list *scripts_queue;
} sentinel;

```

16.1.4 Sentinel masters

Sentinel masters Sentinel

·

· sentinel.c/sentinelRedisInstance


```

int quorum;
// SENTINEL parallel-syncs <master-name> <number>
//
int parallel_syncs;
// SENTINEL failover-timeout <master-name> <ms>
//
mstime_t failover_timeout;
// ...
} sentinelRedisInstance;

```

sentinelRedisInstance.addr

sentinel.c/sentinelAddr IP

```

typedef struct sentinelAddr {
    char *ip;
    int port;
} sentinelAddr;

```

Sentinel masters masters

Sentinel

Sentinel

```

#####
# master1 configure #
#####
sentinel monitor master1 127.0.0.1 6379 2
sentinel down-after-milliseconds master1 30000
sentinel parallel-syncs master1 1
sentinel failover-timeout master1 900000
#####
# master2 configure #

```

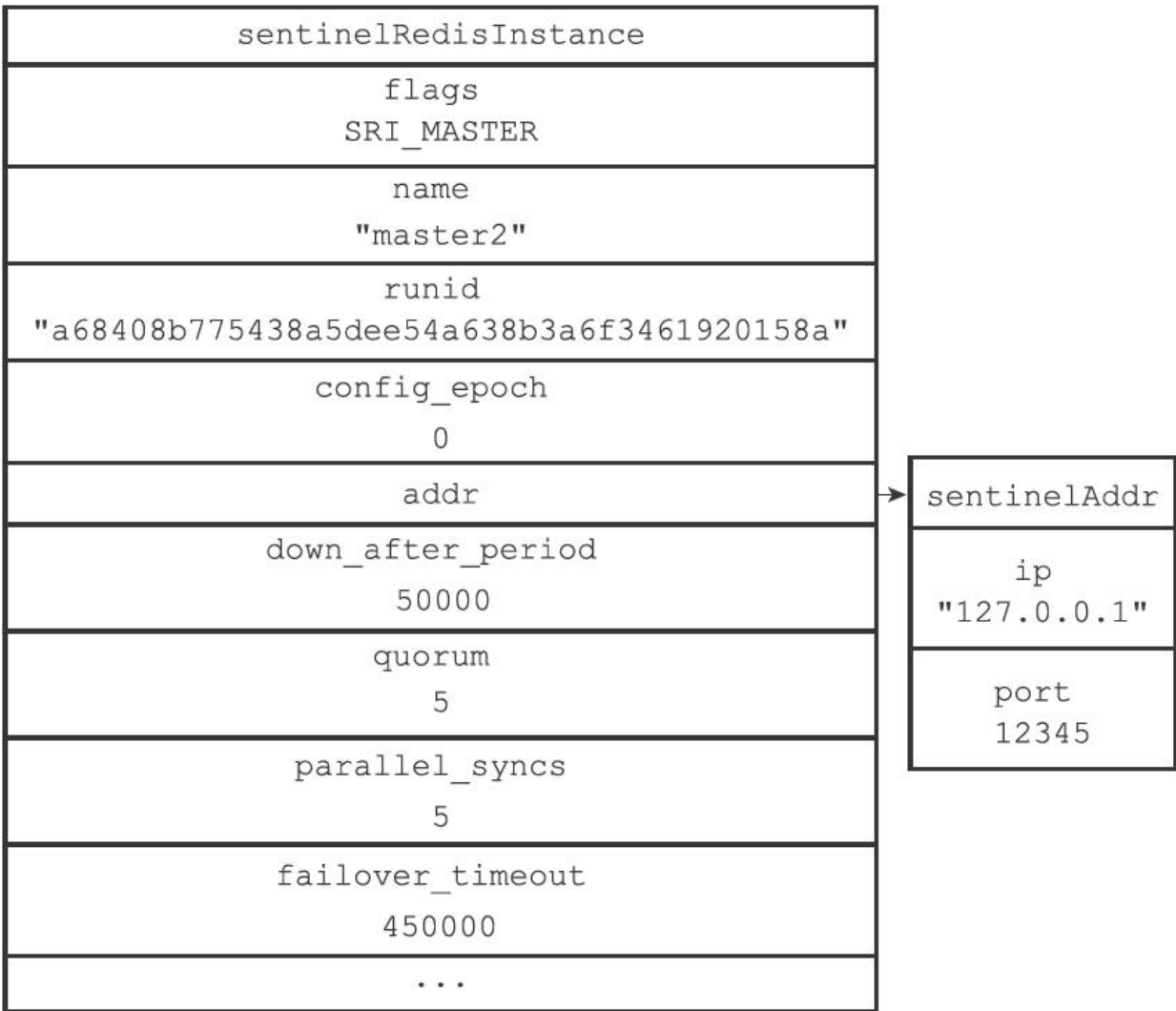



图16-6 master2结构图

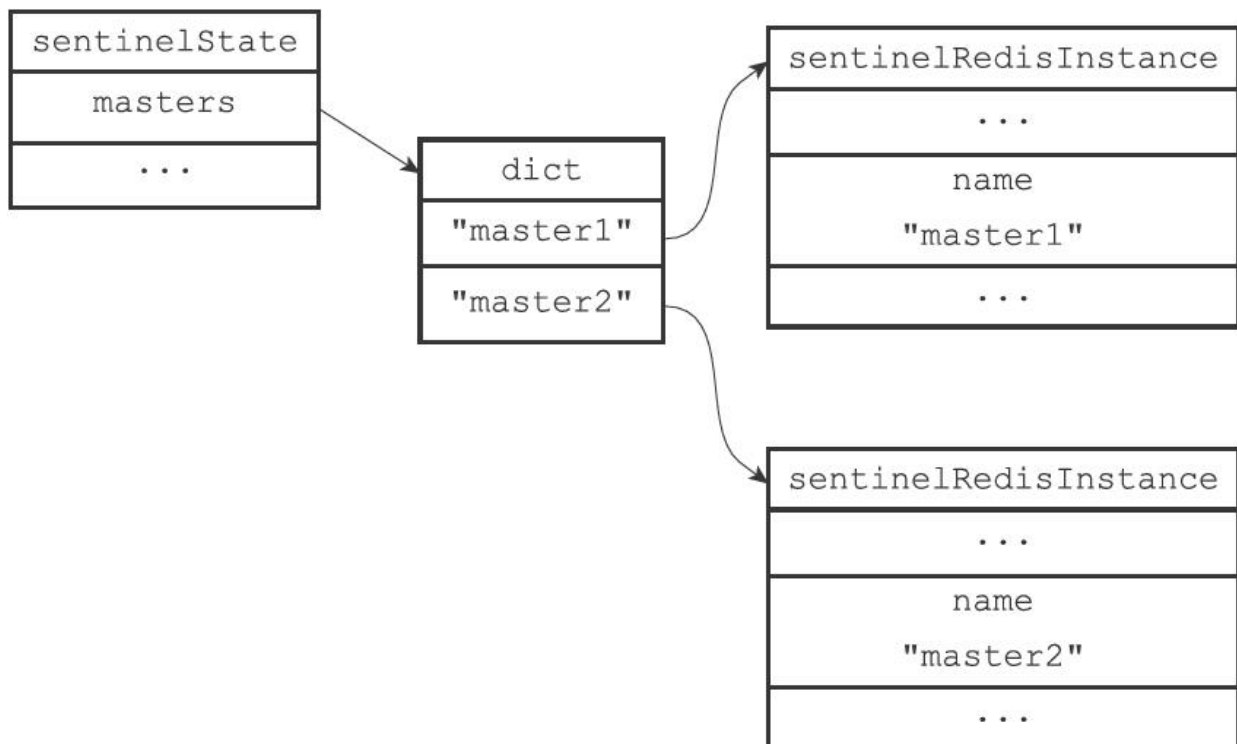


图16-7 Sentinel中的masters

16.1.5 哨兵如何工作

每个Sentinel实例都会定期向每个主实例和从实例发送心跳消息。如果主实例或从实例没有响应，那么Sentinel实例就会认为该实例已经下线。如果主实例下线，那么Sentinel实例就会启动故障转移。

每个Sentinel实例都会定期向每个主实例和从实例发送心跳消息。

· 每个Sentinel实例都会定期向每个主实例和从实例发送心跳消息。

· 每个Sentinel实例都会定期向每个主实例和从实例发送心跳消息。

图16-8 Sentinel的工作流程

16-8 Sentinel

16.2 Sentinel

Sentinel INFO
INFO

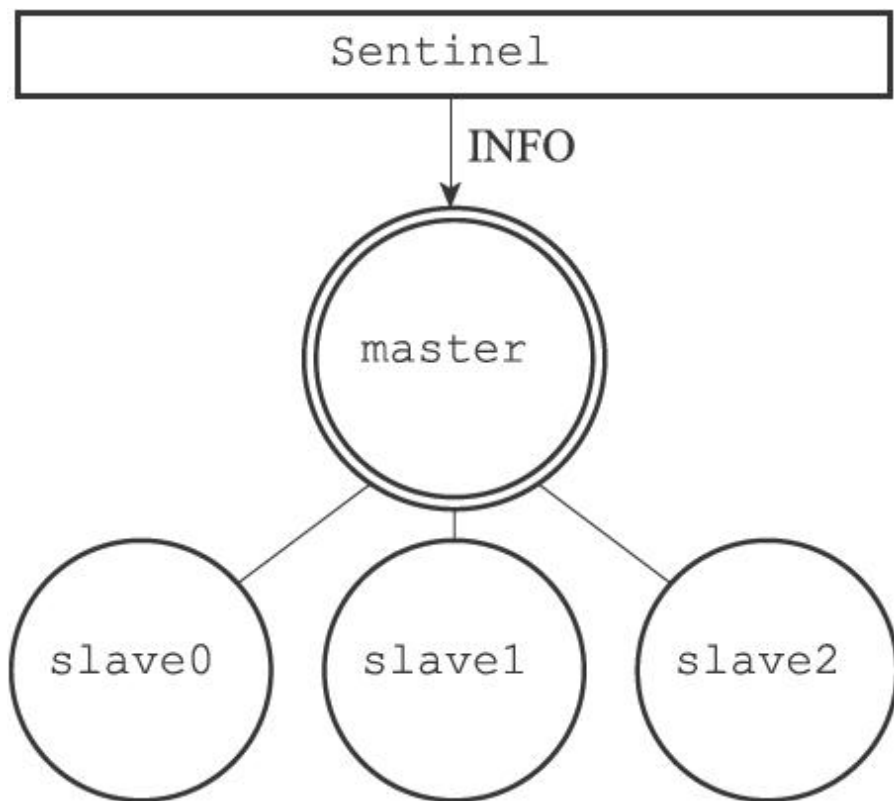


图16-9 Sentinel

图16-9中，master是主节点，slave0、slave1、slave2是从节点。Sentinel是哨兵节点，它通过INFO命令与master和从节点通信。

```
# Server
...
run_id:7611c59dc3a29aa6fa0609f841bb6a1019008a9c
...
# Replication
role:master
...
slave0:ip=127.0.0.1,port=11111,state=online,offset=43,lag=0
slave1:ip=127.0.0.1,port=22222,state=online,offset=43,lag=0
slave2:ip=127.0.0.1,port=33333,state=online,offset=43,lag=0
...
# Other sections
...
```

INFO Sentinel

·run_idIDrole

·"slave"
ip=IPport=IP
Sentinel

run_idroleSentinel
IDIDSentinel
ID

slaves

· Sentinel 的配置文件 ip:port 指定 IP 地址
127.0.0.1 和 11111 指定 Sentinel 的 IP 地址
127.0.0.1:11111

· 指定 127.0.0.1:11111 为 Master
指定 IP 地址 127.0.0.1 和 11111 为 Master

Sentinel 的 INFO 命令返回的信息
slaves

· Sentinel 的配置文件

· Sentinel 的配置文件
slaves

master slave0 slave1 slave2
Sentinel
slaves 16-10

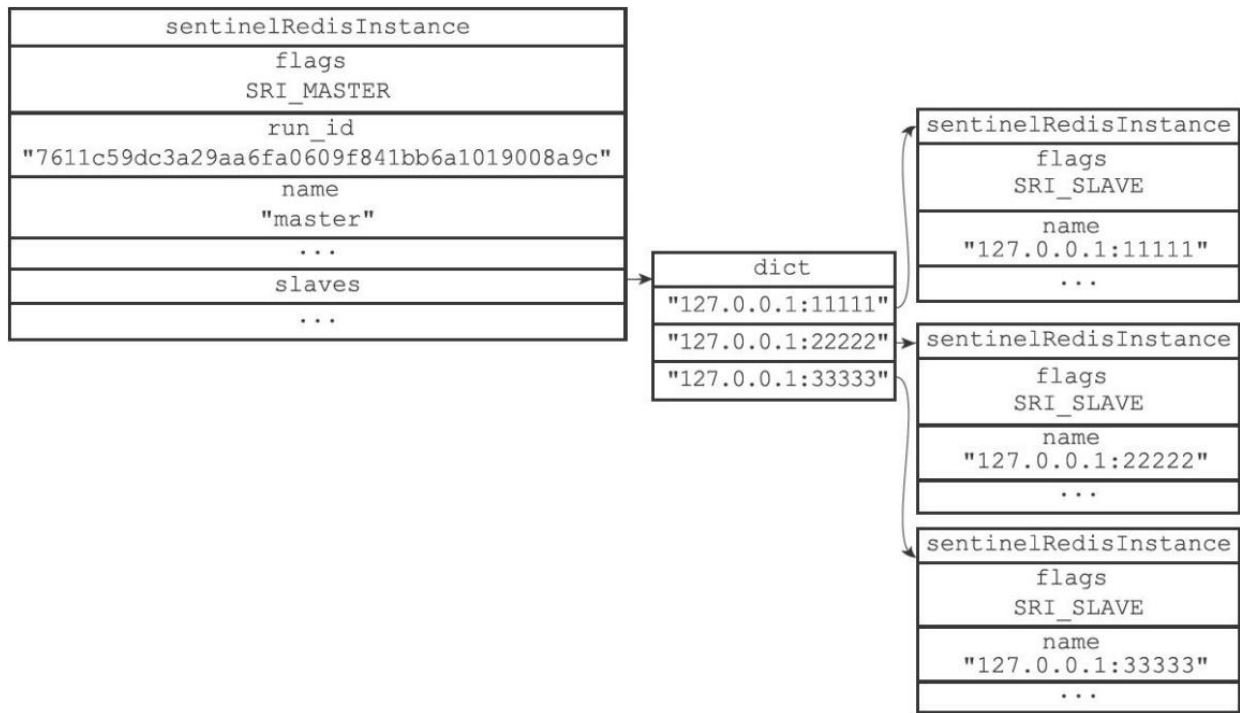


图16-10 哨兵实例结构图

哨兵实例结构图

·哨兵实例的flags字段的SRI_MASTER字段的flags字段的SRI_SLAVE

·哨兵实例的name字段的Sentinel字段的name字段的Sentinel字段的IP字段的IP

16.3 哨兵配置

哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置。

哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置。

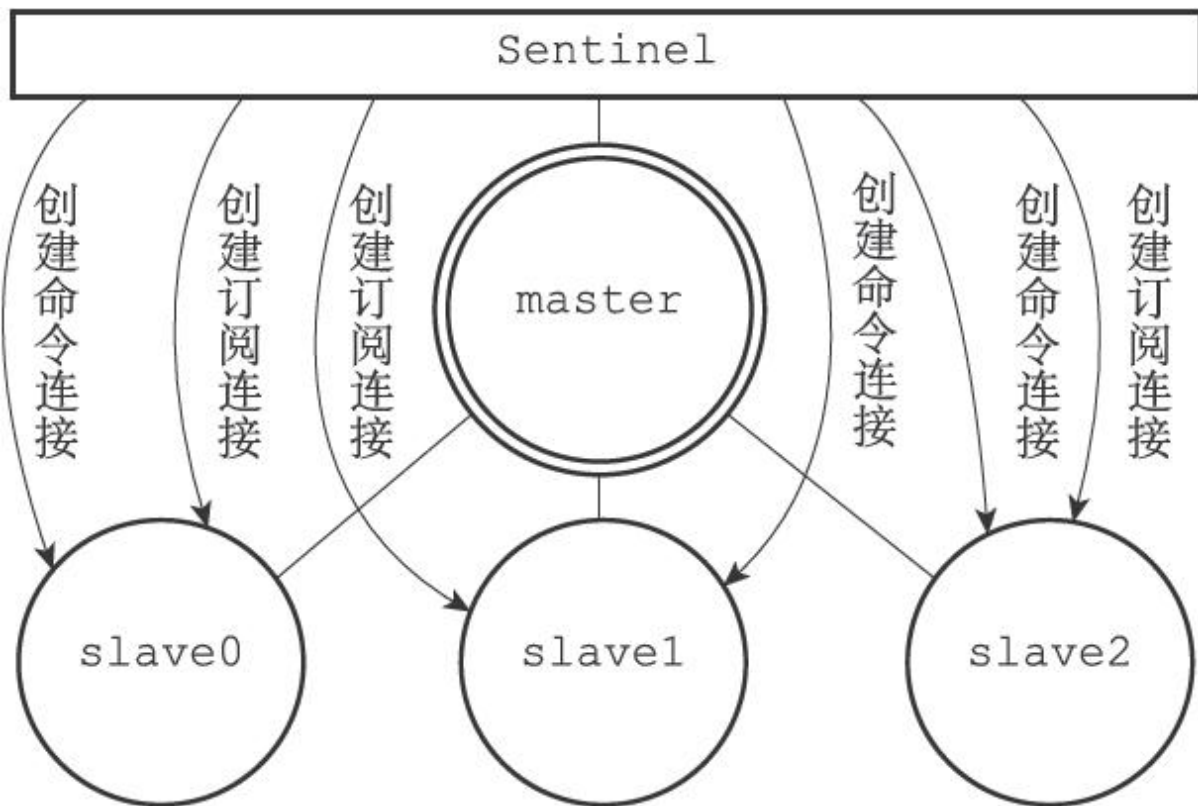


图16-11 Sentinel配置

哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置，哨兵配置文件中配置哨兵配置。

INFO

```
# Server
...
run_id:32be0699dd27b410f7c90dada3a6fab17f97899f
...
# Replication
role:slave
master_host:127.0.0.1
master_port:6379
master_link_status:up
slave_repl_offset:11887
slave_priority:100
# Other sections
...
```

INFO Sentinel

· ID run_id

· role

· IP master_host master_port

· master_link_status

· slave_priority

· slave_repl_offset

Sentinel 16-12 Sentinel

INFO

sentinelRedisInstance
flags SRI_SLAVE
run_id "32be0699dd27b410f7c90dada3a6fab17f97899f"
slave_master_host "127.0.0.1"
slave_master_port 6379
slave_master_link_status SENTINEL_MASTER_LINK_STATUS_UP
slave_repl_offset 11887
slave_priority 100
...

16-12 哨兵配置

16.4 哨兵消息格式

哨兵消息格式如下：

```
PUBLISH __sentinel__:hello "<s_ip>,<s_port>,<s_runid>,<s_epoch>,<m_name>,<m_ip>,<m_port>,<m_epoch>"
```

哨兵消息格式如下：

·s_哨兵消息格式 Sentinel 消息格式 16-2

·m_哨兵消息格式 Sentinel 消息格式 16-3

Sentinel 消息格式

图 16-2 哨兵消息格式

参 数	意 义
s_ip	Sentinel 的 IP 地址
s_port	Sentinel 的端口号
s_runid	Sentinel 的运行 ID
s_epoch	Sentinel 当前的配置纪元（configuration epoch）

图 16-3 哨兵消息格式

参 数	意 义
m_name	主服务器的名字
m_ip	主服务器的 IP 地址
m_port	主服务器的端口号
m_epoch	主服务器当前的配置纪元

配置 Sentinel 的 PUBLISH 命令

```
"127.0.0.1,26379,e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa,0,mymaster,127.0.0.1,6379,0"
```

配置命令

· Sentinel IP 127.0.0.1 端口 26379 ID e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa 纪元 0

· 主节点 mymaster IP 127.0.0.1 端口 6379 纪元 0

16.5 Sentinel 哨兵与 Redis 主从复制

当 Sentinel 检测到主从复制出现问题时，它会通过 Redis 的 SUBSCRIBE 命令订阅主从复制的频道，以便接收主从复制的相关信息。

```
SUBSCRIBE __sentinel__:hello
```

Sentinel 通过 __sentinel__:hello 频道接收主从复制的相关信息，并据此进行故障检测和故障转移。

通过命令连接，Sentinel 向主从复制发送信息；通过订阅连接，Sentinel 从主从复制接收信息。图 16-13 展示了 Sentinel 与主从复制的连接方式。

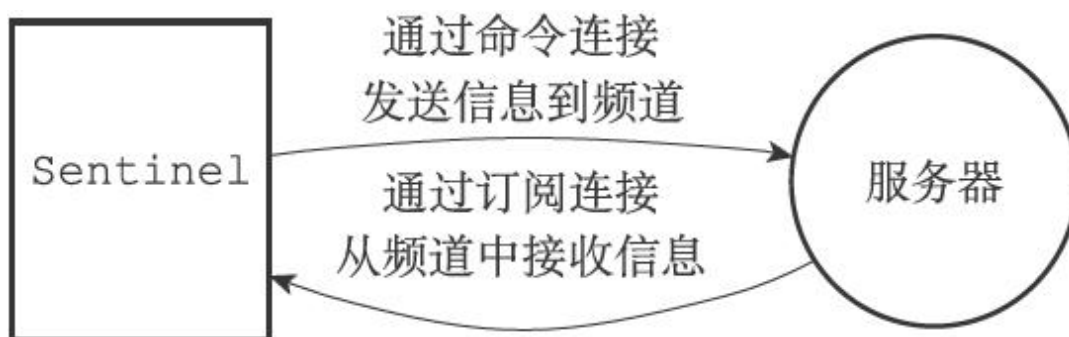
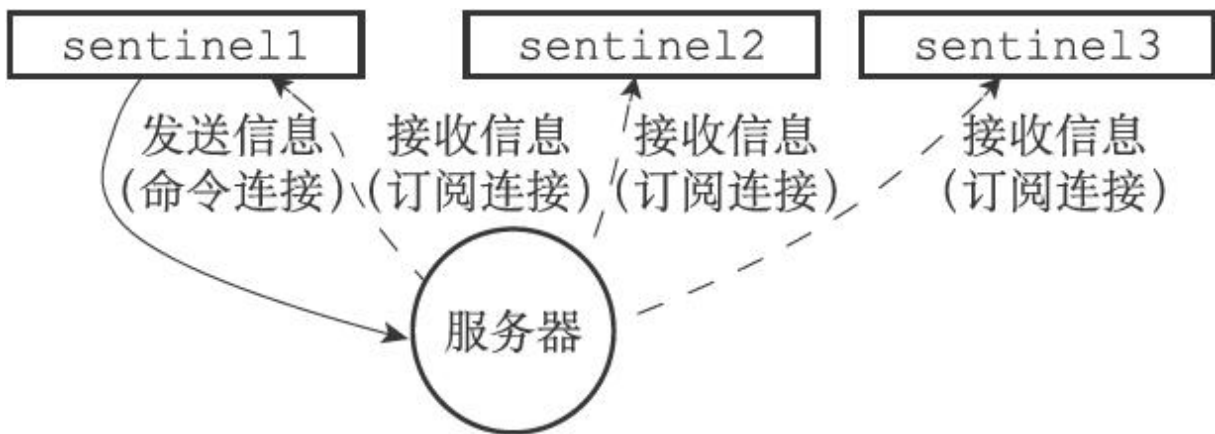


图 16-13 Sentinel 与主从复制的连接方式

当 Sentinel 检测到主从复制出现问题时，它会通过 Redis 的 SUBSCRIBE 命令订阅主从复制的频道，以便接收主从复制的相关信息。

Sentinel

sentinel1 sentinel2 sentinel3 Sentinel
sentinel1 __sentinel__:hello
__sentinel__:hello Sentinel sentinel1
16-14



16-14

Sentinel __sentinel__:hello Sentinel
Sentinel IP Sentinel Sentinel ID

· Sentinel ID Sentinel ID
Sentinel Sentinel

· Sentinel ID Sentinel ID
Sentinel Sentinel

redis.conf 文件中的配置

16.5.1 配置sentinels

Sentinel 守护进程通过 sentinel 配置项指定 Sentinel 守护进程的
主从配置和 Sentinel 守护进程

· sentinel 配置项指定 Sentinel 守护进程的 ip:port 和 IP
127.0.0.1 和 26379 指定 Sentinel 守护进程的 sentinel 配置项
为 "127.0.0.1:26379"

· sentinel 配置项指定 Sentinel 守护进程的
"127.0.0.1:26379" 指定 sentinel 配置项 IP 127.0.0.1 和
26379 指定 Sentinel 守护进程

配置 Sentinel 守护进程的 Sentinel 守护进程的 Sentinel 守护
Sentinel 守护进程的 Sentinel 守护进程的 Sentinel 守护进程的
守护进程的

· Sentinel 守护进程的 Sentinel IP 守护进程的 ID 守护进程的

· 守护进程的 Sentinel 守护进程的 IP 守护进程的

配置 Sentinel 守护进程的 Sentinel 守护进程的 masters
守护进程的 Sentinel 守护进程的

sentinels Sentinel

· Sentinel Sentinel

· Sentinel Sentinel

Sentinel Sentinel Sentinel

sentinels

127.0.0.1:26379 127.0.0.1:26380

127.0.0.1:26381 Sentinel 127.0.0.1:6379

127.0.0.1:26379 Sentinel

```
1) "message"
2) "__sentinel__:hello"
3)
"127.0.0.1,26379,e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa,0,mymaster,127.0.0.1,6379,0"
1) "message"
2) "__sentinel__:hello"
3)
"127.0.0.1,26381,6241bf5cf9bfc8ecd15d6eb6cc3185edfbb24903,0,mymaster,127.0.0.1,6379,0"
1) "message"
2) "__sentinel__:hello"
3)
"127.0.0.1,26380,a9b22fb79ae8fad28e4ea77d20398f77f6b89377,0,mymaster,127.0.0.1,6379,0"
```

Sentinel

· 127.0.0.1:26379

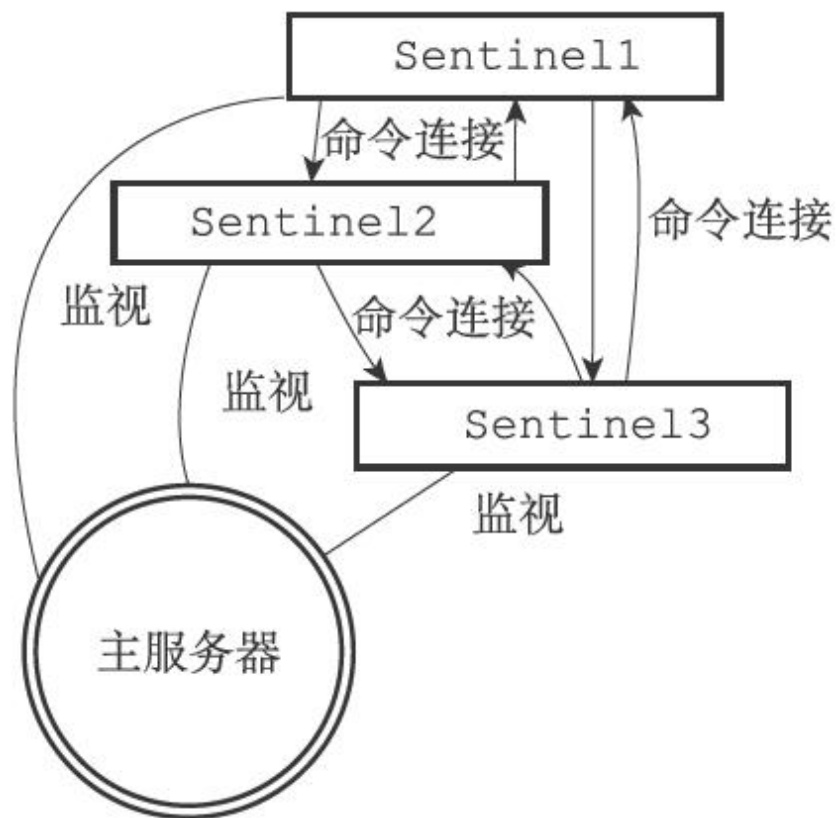


图16-16 Sentinel架构图

图16-16展示了Sentinel的架构图。

Sentinel通过命令连接（命令连接）相互连接。

Sentinel通过命令连接（命令连接）相互连接。Sentinel通过命令连接（命令连接）相互连接。Sentinel通过命令连接（命令连接）相互连接。

Sentinel架构图

Sentinel通过命令连接（命令连接）相互连接。

Sentinel通过命令连接（命令连接）相互连接。Sentinel通过命令连接（命令连接）相互连接。

哨兵是Redis集群中最重要的角色之一，负责监控集群中其他节点的状态。哨兵通过定期发送心跳来检测主节点或从节点是否存活。如果检测到故障，哨兵会启动故障转移流程，将主节点的角色切换到从节点上。

Sentinel是Redis集群中负责监控和管理节点的角色。它通过定期发送心跳来检测主节点或从节点是否存活。如果检测到故障，Sentinel会启动故障转移流程，将主节点的角色切换到从节点上。

16.6 Redis Sentinel

Redis Sentinel 是一个高可用解决方案，它通过多个 Sentinel 实例对 Redis 主从集群进行监控。当主节点发生故障时，Sentinel 会自动将流量切换到从节点，确保服务的连续性。

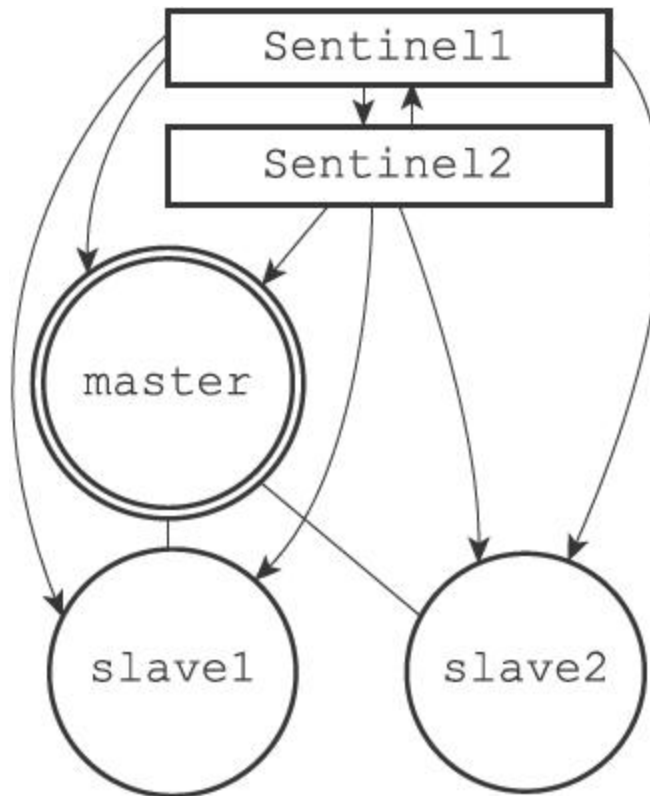


图16-17 Redis Sentinel 配置

在图16-17中，我们配置了两个 Sentinel 实例（Sentinel1 和 Sentinel2）来监控一个 Redis 主从集群。每个 Sentinel 实例都会定期向主节点和从节点发送 PING 命令，以检查它们的健康状态。

· Sentinel1 和 Sentinel2 都会向 master 节点和 slave1 节点发送 PING 命令。

· Sentinel2 → Sentinel1 → master → slave1 → slave2 →
PING →

→ PING →

· → + PONG → -LOADING → -MASTERDOWN →
→

· → + PONG → -LOADING → -MASTERDOWN →
→

Sentinel → down-after-milliseconds → Sentinel →
→ down-after-milliseconds →
Sentinel → Sentinel → flags →
→ SRI_S_DOWN →

→ 16-17 → Sentinel1 → down-after-
milliseconds → 50000 → master → 50000 →
Sentinel1 → Sentinel1 → master → master →
→ flags → SRI_S_DOWN → 16-18 →


```

    master 10000 Sentinel2 master
    Sentinel1 master master 50000
    Sentinel1 Sentinel2 master

```

16.7 哨兵主观下线

哨兵主观下线是指哨兵判断主服务器不可用，并通知其他哨兵。哨兵主观下线的原因有很多，比如主服务器宕机、主服务器配置错误、主服务器与哨兵之间的网络问题等。哨兵主观下线后，其他哨兵会根据哨兵的反馈，对主服务器的状态进行重新评估。如果哨兵判断主服务器确实不可用，那么哨兵就会将主服务器的状态设置为“主观下线”，并通知其他哨兵。其他哨兵收到通知后，会根据哨兵的反馈，对主服务器的状态进行重新评估。如果哨兵判断主服务器确实不可用，那么哨兵就会将主服务器的状态设置为“主观下线”，并通知其他哨兵。其他哨兵收到通知后，会根据哨兵的反馈，对主服务器的状态进行重新评估。

16.7.1 哨兵SENTINEL is-master-down-by-addr

哨兵命令

```
SENTINEL is-master-down-by-addr <ip> <port> <current_epoch> <runid>
```

哨兵命令哨兵主观下线主服务器16-4

图16-4 SENTINEL is-master-down-by-addr命令示意图

参 数	意 义
ip	被 Sentinel 判断为主观下线的主服务器的 IP 地址
port	被 Sentinel 判断为主观下线的主服务器的端口号
current_epoch	Sentinel 当前的配置纪元，用于选举领头 Sentinel，详细作用将在下一节说明
runid	可以是 * 符号或者 Sentinel 的运行 ID：* 符号代表命令仅仅用于检测主服务器的客观下线状态，而 Sentinel 的运行 ID 则用于选举领头 Sentinel，详细作用将在下一节说明

目标 Sentinel 的 IP 127.0.0.1 的端口 6379 的 Sentinel 的 Sentinel 的 Sentinel

```
SENTINEL is-master-down-by-addr 127.0.0.1 6379 0 *
```

16.7.2 SENTINEL is-master-down-by-addr

目标 Sentinel 的 Sentinel 的 Sentinel 的 Sentinel 的 Sentinel is-master-down-by 的 Sentinel 的 IP 的 Sentinel 的 Multi Bulk 的 SENTINEL is-master-down-by

- 1) <down_state>
- 2) <leader_runid>
- 3) <leader_epoch>

16-5

16-5 SENTINEL is-master-down-by-addr

参 数	意 义
down_state	返回目标 Sentinel 对主服务器的检查结果，1 代表主服务器已下线，0 代表主服务器未下线
leader_runid	可以是 * 符号或者目标 Sentinel 的局部领头 Sentinel 的运行 ID；* 符号代表命令仅仅用于检测主服务器的下线状态，而局部领头 Sentinel 的运行 ID 则用于选举领头 Sentinel，详细作用将在下一节说明
leader_epoch	目标 Sentinel 的局部领头 Sentinel 的配置纪元，用于选举领头 Sentinel，详细作用将在下一节说明。仅在 leader_runid 的值不为 * 时有效，如果 leader_runid 的值为 *，那么 leader_epoch 总为 0

redis sentinel 1 redis sentinel 2
quorum redis sentinel redis sentinel
Sentinel redis

sentinel monitor master 127.0.0.1 6379 2

redis sentinel redis sentinel
redis sentinel redis sentinel redis sentinel

sentinel monitor master 127.0.0.1 6379 5

redis sentinel redis sentinel
Sentinel redis

redis sentinel

redis sentinel redis sentinel
redis sentinel redis sentinel
redis sentinel redis sentinel1 redis

sentinel monitor master 127.0.0.1 6379 2

```

00000000 Sentinel0000 Sentinel000000000000 Sentinel1000
0000000000000000

```

Sentinel2

```
sentinel monitor master 127.0.0.1 6379 5
```

```

Sentinel1Sentinel2

```

16.8 Sentinel

```

00000000000000000000000000000000Sentinel0000000000
00Sentinel00000Sentinel000000000000000000000000

```

Redis Sentinel

```
· Sentinel Sentinel  
Sentinel Sentinel
```

```
· Sentinel Sentinel
configuration epoch

```

```
· Sentinel Sentinel Sentinel  
Sentinel
```

```
· Sentinel Sentinel Sentinel
Sentinel
```

```
· Sentinel Sentinel Sentinel Sentinel Sentinel
SENTINEL is-master-down-by-addr runid *
 Sentinel ID Sentinel Sentinel
Sentinel
```


· Sentinel 主哨兵 Sentinel 副哨兵 Sentinel 哨兵
哨兵 Sentinel 哨兵 Sentinel 哨兵 Sentinel 哨兵
Sentinel 哨兵

· 哨兵 Sentinel 哨兵 SENTINEL is-master-down-by-addr 哨兵
哨兵 Sentinel 哨兵 leader_runid 哨兵 leader_epoch 哨兵
哨兵 Sentinel 哨兵 Sentinel 哨兵 ID 哨兵

· Sentinel 哨兵 Sentinel 哨兵
leader_epoch 哨兵 Sentinel 哨兵
哨兵 leader_runid 哨兵 leader_runid 哨兵 Sentinel 哨兵 ID 哨兵
哨兵 Sentinel 哨兵 Sentinel 哨兵 Sentinel

· 哨兵 Sentinel 哨兵 Sentinel 哨兵 Sentinel
Sentinel 哨兵 Sentinel 哨兵 10 Sentinel 哨兵 Sentinel
哨兵 哨兵 $10/2+1=6$ Sentinel 哨兵 Sentinel 哨兵
哨兵 Sentinel 哨兵 Sentinel

· 哨兵 Sentinel 哨兵 Sentinel 哨兵 Sentinel
哨兵 Sentinel 哨兵 Sentinel

· 哨兵 Sentinel 哨兵 Sentinel 哨兵
哨兵 Sentinel

哨兵通过定期发送心跳来检测主服务器的存活。如果哨兵发现主服务器在指定的时间内没有响应，那么哨兵就会认为主服务器已经下线，并会发送消息给其他哨兵，让它们也进入故障转移状态。

哨兵通过定期发送心跳来检测主服务器的存活。如果哨兵发现主服务器在指定的时间内没有响应，那么哨兵就会认为主服务器已经下线，并会发送消息给其他哨兵，让它们也进入故障转移状态。

SENTINEL is-master-down-by-addr 哨兵通过定期发送心跳来检测主服务器的存活。如果哨兵发现主服务器在指定的时间内没有响应，那么哨兵就会认为主服务器已经下线，并会发送消息给其他哨兵，让它们也进入故障转移状态。

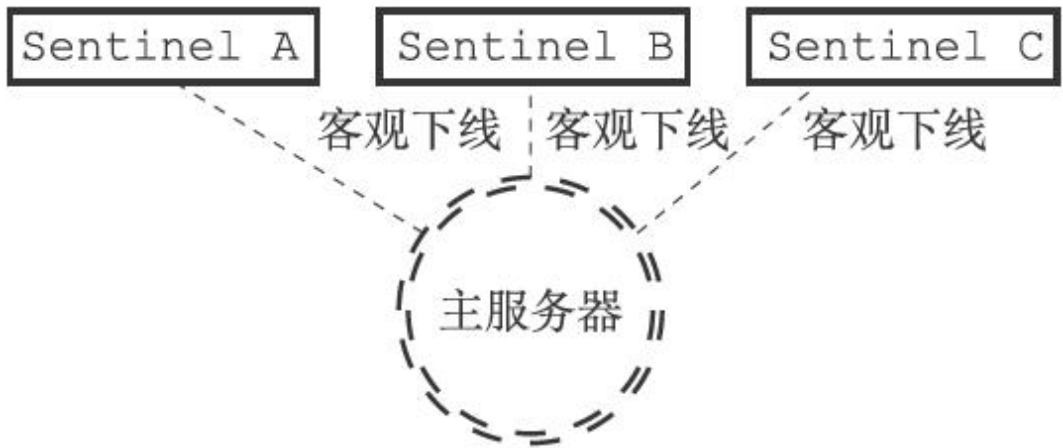
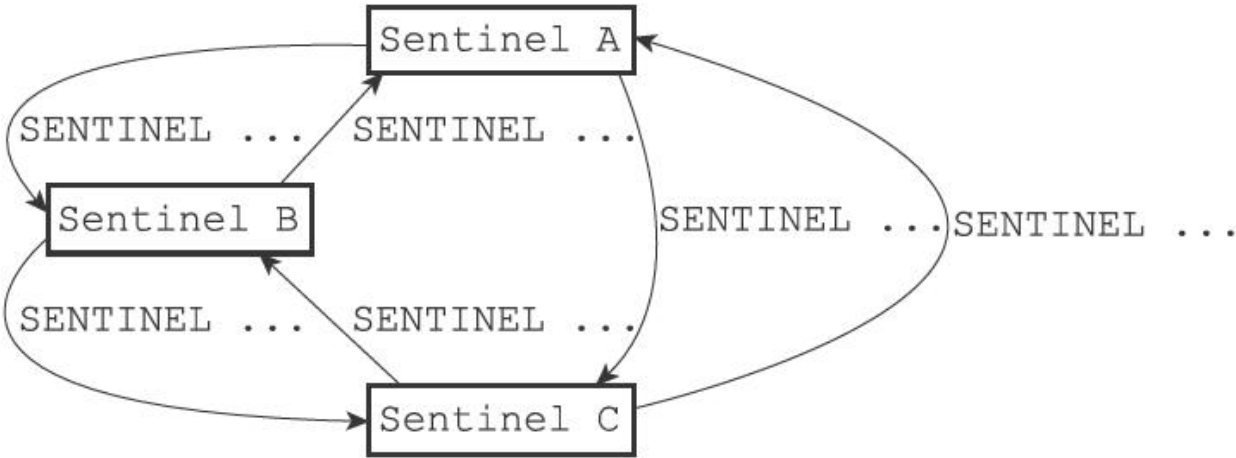


图16-20 哨兵通过定期发送心跳来检测主服务器的存活

哨兵通过定期发送心跳来检测主服务器的存活。如果哨兵发现主服务器在指定的时间内没有响应，那么哨兵就会认为主服务器已经下线，并会发送消息给其他哨兵，让它们也进入故障转移状态。

SENTINEL is-master-down-by-addr 哨兵通过定期发送心跳来检测主服务器的存活。如果哨兵发现主服务器在指定的时间内没有响应，那么哨兵就会认为主服务器已经下线，并会发送消息给其他哨兵，让它们也进入故障转移状态。



16-21 Sentinel Sentinel

SENTINEL is-master-down-by-addr
Sentinel ID

```
SENTINEL is-master-down-by-addr 127.0.0.1 6379 0  
e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa
```

Sentinel Sentinel ID
e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa Sentinel
Sentinel

-
- 1) 1
 - 2) e955b4c85598ef5b5f055bc7ebfd5e828dbed4fa
 - 3) 0
-

Sentinel Sentinel
Sentinel

Sentinel SENTINEL is-master-
down-by-addr Sentinel Sentinel
Sentinel Sentinel

16.9 哨兵

哨兵是 Redis 的高可用解决方案。哨兵通过监控 Redis 主从实例，当主实例发生故障时，哨兵会自动将主实例的角色切换到从实例，从而实现高可用。

1. 哨兵通过配置文件中指定的地址，定期向主从实例发送 PING 命令，以检查实例是否存活。

2. 哨兵通过配置文件中指定的地址，定期向主从实例发送 INFO 命令，以获取实例的元数据。

3. 哨兵通过配置文件中指定的地址，定期向主从实例发送 SETBIT 命令，以设置实例的故障标志。当故障标志被设置时，哨兵会自动将主实例的角色切换到从实例。

16.9.1 哨兵配置

哨兵的配置主要通过配置文件中的 `sentinel` 命令来配置。配置项包括哨兵的地址、主从实例的地址、哨兵的故障标志等。

哨兵配置项
<code>sentinel monitor mymaster 127.0.0.1 6379 2</code> 哨兵配置项
<code>1 127.0.0.1 6379</code> 哨兵

```
2哨兵通过配置文件中配置的哨兵地址向Sentinel的INFO命令来检测主从复制状态
哨兵通过检测主从复制状态来判断主从复制是否成功
```

```
3哨兵通过配置文件中配置的哨兵地址向down-after-milliseconds*10来检测主从复制状态
哨兵通过down-after-milliseconds来检测主从复制是否成功
哨兵通过down-after-milliseconds*10来检测主从复制是否成功
哨兵通过检测主从复制状态来判断主从复制是否成功
```

```
哨兵通过Sentinel来检测主从复制是否成功
哨兵通过检测主从复制状态来判断主从复制是否成功
```

```
哨兵通过配置文件中配置的哨兵地址向Sentinel来检测主从复制状态
哨兵通过检测主从复制状态来判断主从复制是否成功
哨兵通过检测主从复制状态来判断主从复制是否成功
```

```
哨兵通过配置文件中配置的哨兵地址向Sentinel来检测主从复制状态
哨兵通过检测主从复制状态来判断主从复制是否成功
```

16-22哨兵通过配置文件中配置的哨兵地址向Sentinel来检测主从复制状态server2
SLAVEOF no one哨兵通过检测主从复制状态来判断主从复制是否成功

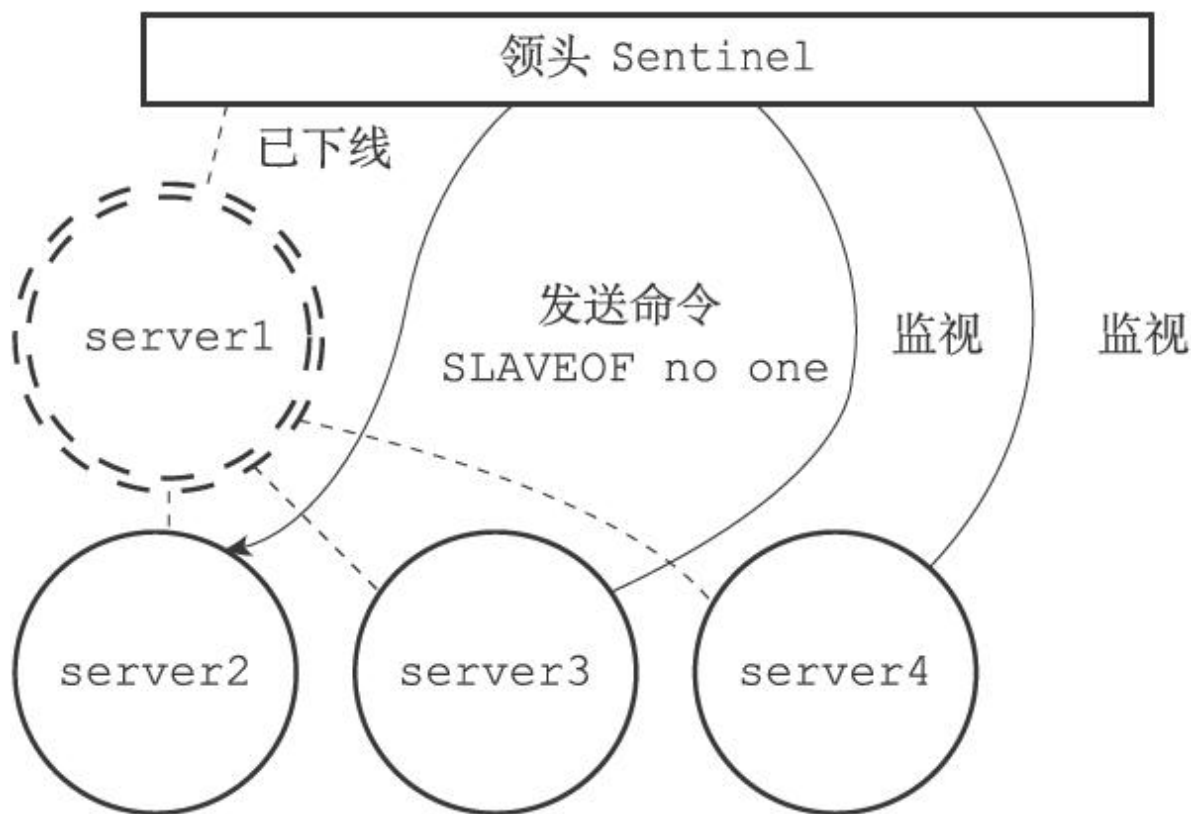


图16-22 配置server2为从节点

在配置文件中添加以下配置，将server2配置为从节点，并指定其主节点为server1。

```

# Replication
role:slave
...
# Other sections
...

```

图16-22展示了配置server2为从节点的过程。通过配置Sentinel，可以监控主节点server1，并在其故障时自动将server2提升为主节点。

```

# Replication
role:slave
...
# Other sections
...

```

哨兵

```
# Replication
role:master
...
# Other sections
...
```

哨兵哨兵Sentinel哨兵server2哨兵哨兵哨兵哨兵哨兵

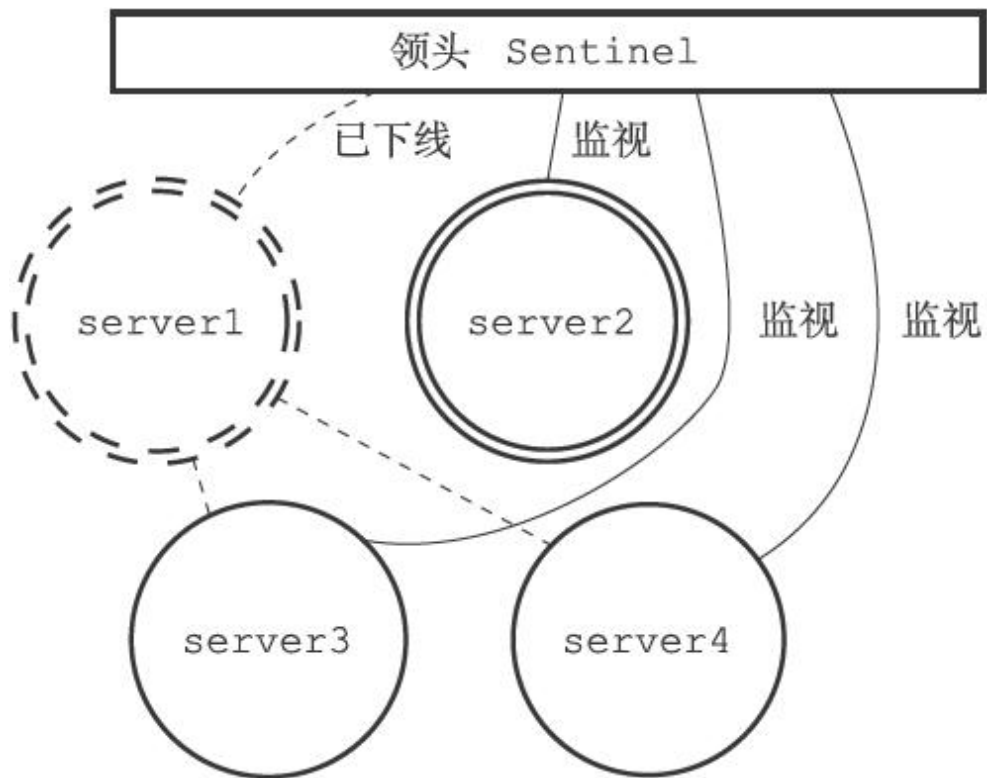


图16-23 server2哨兵哨兵哨兵哨兵哨兵

图16-23哨兵server2哨兵哨兵哨兵哨兵哨兵哨兵Sentinel哨兵哨兵

16.9.2 哨兵哨兵哨兵哨兵哨兵

哨兵通过定期发送 Sentinel 命令来检测主服务器是否存活。如果主服务器没有响应，哨兵就会认为主服务器已经下线，并会向其他哨兵发送 SLAVEOF 命令，让其他哨兵成为主服务器的副本。

图16-24展示了哨兵如何检测主服务器是否存活，并如何向其他哨兵发送 SLAVEOF 命令，让其他哨兵成为主服务器的副本。

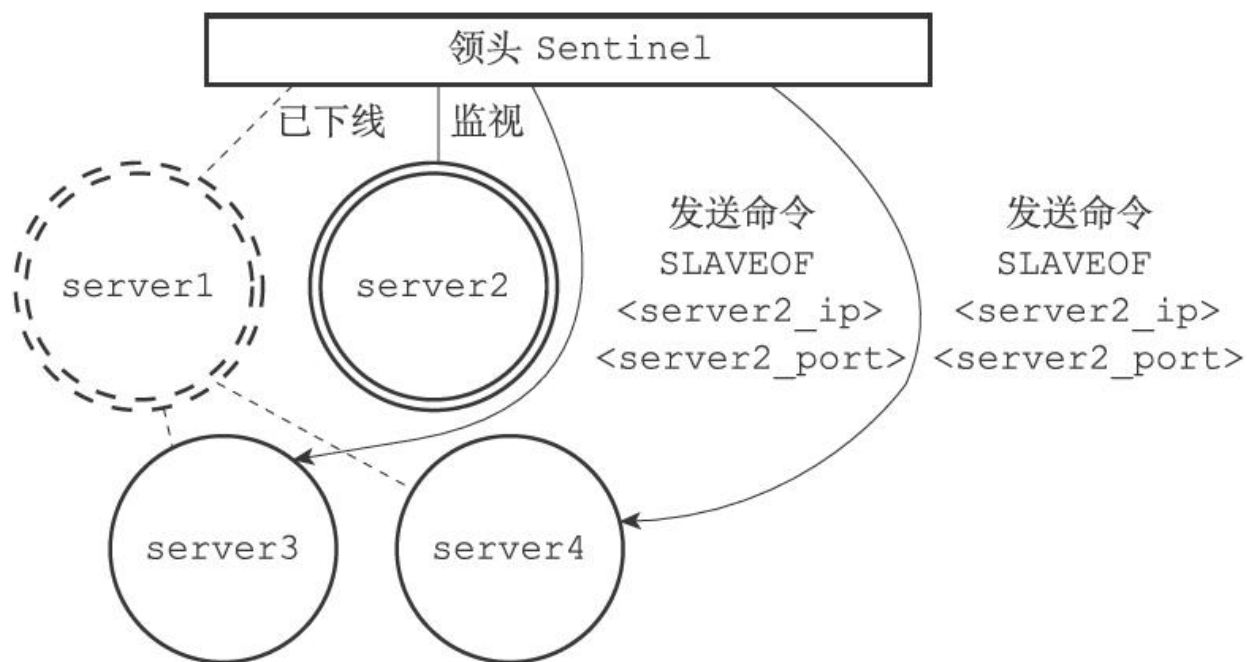


图16-24 哨兵检测主服务器是否存活

图16-25展示了哨兵如何向其他哨兵发送 SLAVEOF 命令，让其他哨兵成为主服务器的副本。图中，哨兵向 server3 和 server4 发送了 SLAVEOF 命令，让它们成为 server2 的副本。

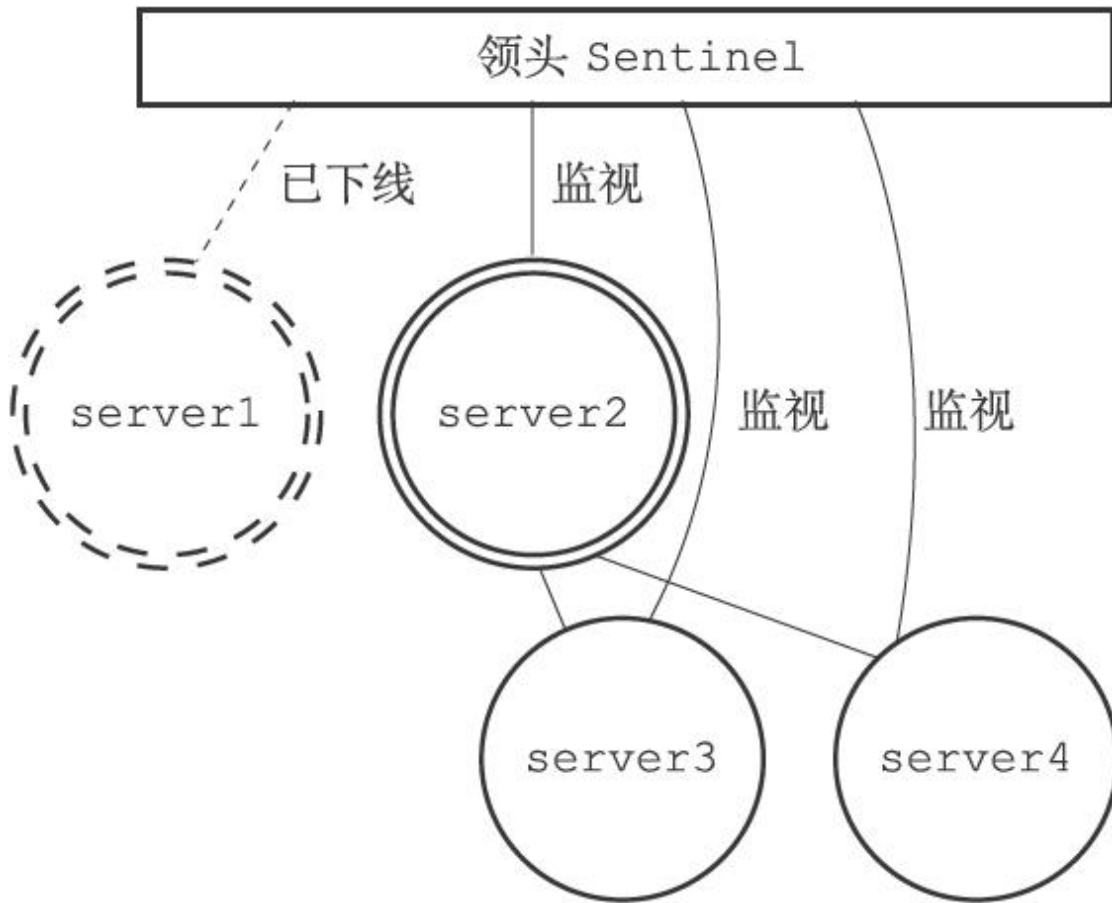


图16-25 server3、server4、server2的监视配置

16.9.3 配置哨兵节点

配置哨兵节点需要配置哨兵节点的配置文件，配置文件的路径为/etc/redis/sentinel.conf，如图16-26所示。配置哨兵节点需要配置哨兵节点的配置文件，配置文件的路径为/etc/redis/sentinel.conf，如图16-26所示。

配置哨兵节点需要配置哨兵节点的配置文件，配置文件的路径为/etc/redis/sentinel.conf，如图16-26所示。配置哨兵节点需要配置哨兵节点的配置文件，配置文件的路径为/etc/redis/sentinel.conf，如图16-26所示。

图16-27展示了server1和server2的配置文件内容。

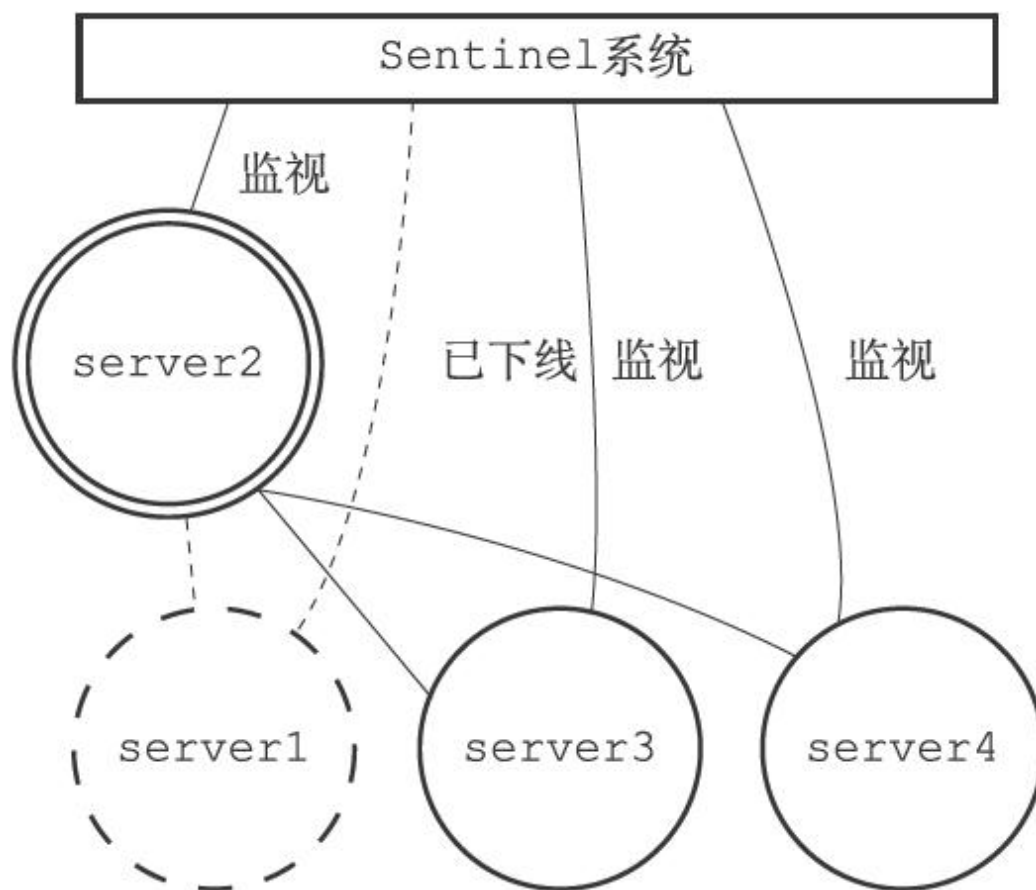


图16-26 server1下线后的哨兵系统

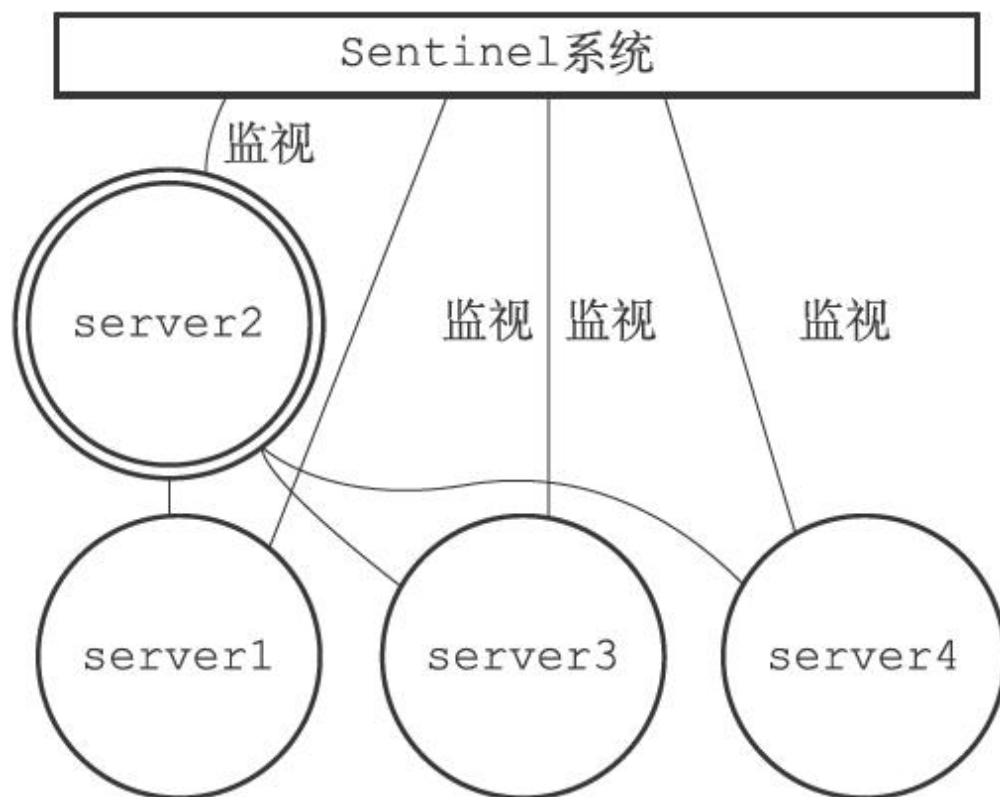


图16-27 server1和server2的复制关系

16.10 哨兵

· Sentinel 哨兵节点通过 Redis 哨兵节点来发现 Redis 主节点和从节点

· Sentinel 哨兵节点通过 Redis 哨兵节点来发现 Redis 主节点和从节点，并且通过 Redis 哨兵节点来发现 Redis 主节点和从节点

· Sentinel 哨兵节点通过 INFO 命令来发现 Redis 主节点和从节点

· 哨兵节点 Sentinel 哨兵节点通过 INFO 命令来发现 Redis 主节点和从节点，并且通过 Redis 哨兵节点来发现 Redis 主节点和从节点

· 哨兵节点 Sentinel 哨兵节点通过 __sentinel__:hello 命令来发现 Redis 主节点和从节点

· 哨兵节点 Sentinel 哨兵节点通过 __sentinel__:hello 命令来发现 Redis 主节点和从节点

· Sentinel 哨兵节点互相之间互相 ping Sentinel 哨兵节点互相之间互相 ping

· Sentinel 哨兵节点互相之间互相 ping Sentinel 哨兵节点互相之间互相 ping
哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping
哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping

· Sentinel 哨兵节点互相之间互相 ping Sentinel 哨兵节点互相之间互相 ping
哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping

· Sentinel 哨兵节点互相之间互相 ping Sentinel 哨兵节点互相之间互相 ping
哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping 哨兵节点互相 ping

16.11 哨兵

Sentinel哨兵 Sentinel哨兵 Raft分布式一致性算法
Raft分布式一致性算法“Raft”
http://v.youku.com/v_show/id_XNjQxOTk5MTk2.html Raft
分布式

17 题

Redis 主从复制 Redis 集群 Redis 分片 sharding 数据库

Redis 数据库

17.1 Redis

Redis 是一个开源的分布式数据库，它支持多种数据类型，如字符串、哈希、列表、集合等。Redis 的架构非常简单，它由一个主节点和多个从节点组成。主节点负责处理所有的读写请求，而从节点则负责复制主节点的数据，以实现高可用性和数据冗余。

Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。

```
CLUSTER MEET <ip> <port>
```

Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。

Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。

```
$ redis-cli -c -p 7000
127.0.0.1:7000> CLUSTER NODES
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected
```

Redis 集群的搭建和配置相对简单，但需要掌握一些基本的命令和配置项。

```
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7001
OK
127.0.0.1:7000> CLUSTER NODES
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master - 0
1388204746210 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected
```

redis 7000 redis 7002 redis 7000 redis 7001 redis

redis

```
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7002
OK
127.0.0.1:7000> CLUSTER NODES
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master - 0
1388204848376 0 connected
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master - 0
1388204847977 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected
```

redis 7000 redis 7001 redis 7002 redis 17-1 redis 17-5 redis

redis



redis 17-1 redis

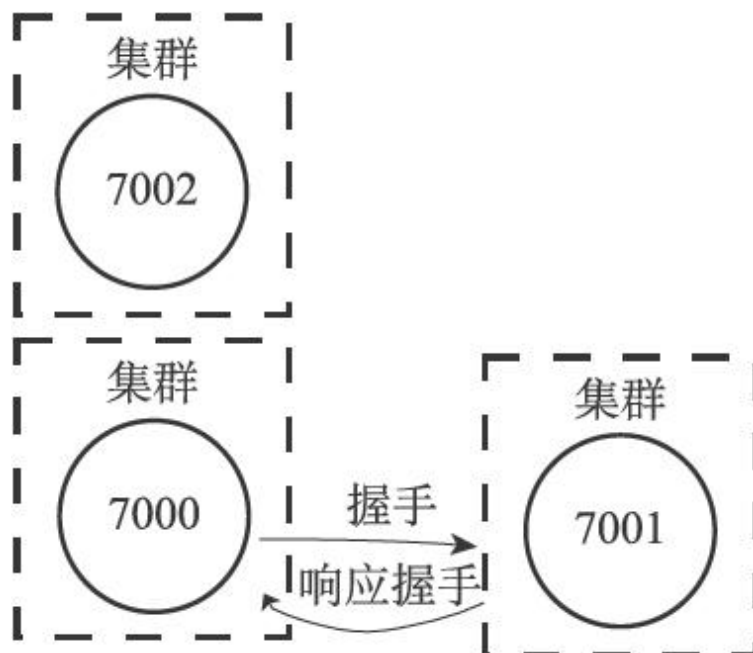


图17-2 节点7000与7001握手

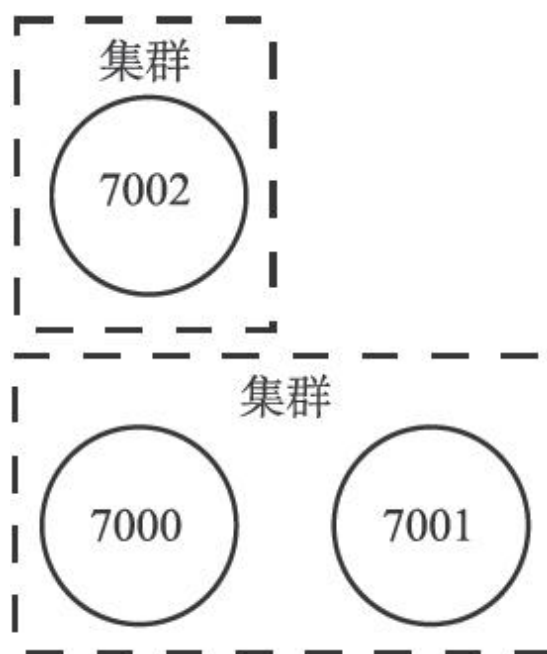


图17-3 节点7000、7001和7002组成一个集群

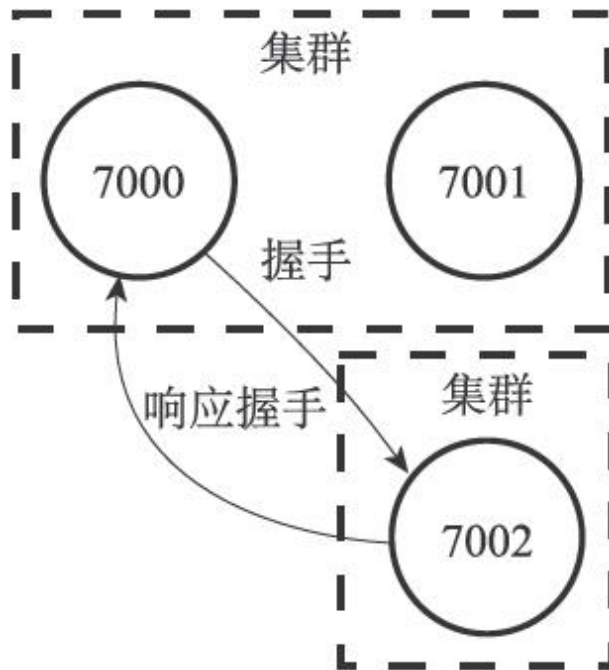


图17-4 节点7000与节点7002握手

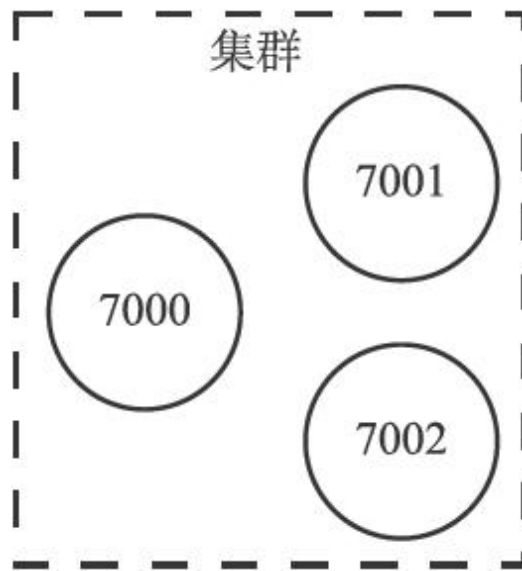


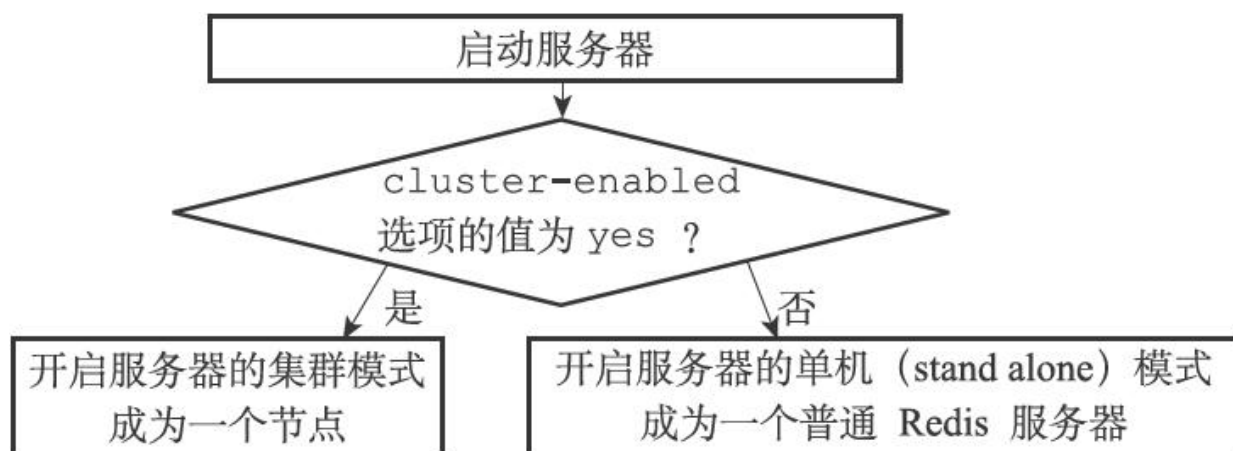
图17-5 节点7000、7001和7002在同一个集群中

当集群中的节点收到来自其他集群的节点发来的CLUSTER MEET消息时

会执行以下操作：

17.1.1 □□□□

```
redis-cluster-enabled yes 17-6
```



□17-6 □□□□□□□□□□□□□□

Redis

- 数据库持久化
- 数据库持久化
 - serverCron 数据库持久化
 - clusterCron 数据库持久化
 - Gossip 数据库持久化
- 数据库持久化
- 数据库持久化
 - RDB 数据库持久化
 - AOF 数据库持久化

- 发布和订阅消息 PUBLISH 和 SUBSCRIBE

- 消息队列

- Lua 脚本

redisServer 和 redisClient

cluster.h/clusterNode cluster.h/clusterLink

cluster.h/clusterState

17.1.2 集群

clusterNode

IP

clusterNode

clusterNode

```
struct clusterNode {  
    //  
    mstime_t ctime;  
    //  
    40  
    //  
    68eef66df23420a5862208ef5b1a7005b806f2ff  
    char name[REDIS_CLUSTER_NAMELEN];  
    //  
}
```


redisClient[] clusterLink[]

redisClient[] clusterLink[]
[] redisClient[] clusterLink
[]

clusterState[]
[]

```
typedef struct clusterState {  
    //  
    clusterNode *myself;  
    //  
    uint64_t currentEpoch;  
    //  
    int state;  
    //  
    int size;  
    //  
    myself  
    //  
    clusterNode  
    dict *nodes;  
    // ...  
} clusterState;
```

redis 7000 7001 7002 redis 17-7 redis 7000
clusterState redis 7000 redis
redis clusterNode redis

· redis currentEpoch redis 0 redis 0

· redis size redis 0 redis state redis
REDIS_CLUSTER_FAIL redis

· redis nodes redis clusterNode redis
redis myself redis 7000 clusterNode redis
redis 7001 redis 7002 clusterNode redis 7000 redis
redis

· redis clusterNode redis flags redis REDIS_NODE_MASTER redis
redis

redis 7001 redis 7002 redis clusterState redis

· redis 7001 redis clusterState redis myself redis 7001
clusterNode redis 7000 redis 7002 redis

· redis 7002 redis clusterState redis myself redis 7002
clusterNode redis 7000 redis 7001 redis

clusterState
myself
currentEpoch
0
state
REDIS_CLUSTER_FAIL
size
0
nodes
...

nodes
"5154...2939"
"68ee...f2ff"
"9dfb...5c26"

clusterNode
name
"5154...2939"
flags
REDIS_NODE_MASTER
configEpoch
0
ip
"127.0.0.1"
port
7000
...

clusterNode
name
"68ee...f2ff"
flags
REDIS_NODE_MASTER
configEpoch
0
ip
"127.0.0.1"
port
7001
...

clusterNode
name
"9dfb...5c26"
flags
REDIS_NODE_MASTER
configEpoch
0
ip
"127.0.0.1"
port
7002
...

图17-7 每隔7000ms发送clusterState消息

17.1.3 CLUSTER MEET消息

节点A向节点B发送CLUSTER MEET消息，消息中包含节点A的IP地址和端口号，以及节点A的clusterState消息。

CLUSTER MEET <ip> <port>

节点A和节点B通过handshake消息进行握手，握手成功后，节点A向节点B发送CLUSTER MEET消息。

1. 节点A向节点B发送clusterNode消息，消息中包含节点A的clusterState消息。

2. 节点A向节点B发送CLUSTER MEET消息，消息中包含节点A的IP地址和端口号，以及节点A的clusterState消息。

3. 节点B向节点A发送MEET消息，消息中包含节点B的clusterNode消息。

4. 节点B向节点A发送PONG消息。

5. 节点A向节点B发送PONG消息，消息中包含节点A的clusterState消息。

6 节点 A 向 B 发送 PING 消息

7 节点 B 向 A 发送 PING 消息，节点 A 向 B 发送 PONG 消息

图 17-8 节点 A 和 B 的交互

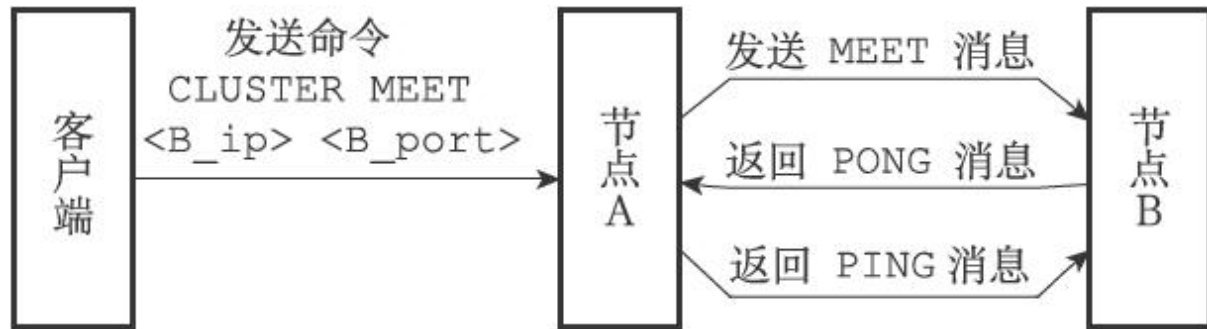


图 17-8 节点 A 和 B 的交互

节点 A 向 B 发送 Gossip 消息，节点 B 向 A 发送 Gossip 消息

17.2 配置

Redis 默认配置文件中配置了 16384 个 slot，slot 总数为 16384，每个 slot 的权重为 0。

Redis 默认配置文件中配置了 16384 个 slot，每个 slot 的权重为 0，配置文件中配置了 ok 和 fail 两个配置项。

Redis 默认配置文件中配置了 CLUSTER MEET 命令，用于配置集群节点，配置文件中配置了 7000、7001、7002 三个配置项。

```
127.0.0.1:7000> CLUSTER INFO
cluster_state:fail
cluster_slots_assigned:0
cluster_slots_ok:0
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:3
cluster_size:0
cluster_current_epoch:0
cluster_stats_messages_sent:110
cluster_stats_messages_received:28
```

Redis 默认配置文件中配置了 CLUSTER ADDSLOTS 命令，用于配置集群节点，配置文件中配置了 assign 和 fail 两个配置项。

```
CLUSTER ADDSLOTS <slot> [slot ...]
```

050007000

```
127.0.0.1:7000> CLUSTER ADDSLOTS 0 1 2 3 4 ... 5000
OK
127.0.0.1:7000> CLUSTER NODES
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master - 0
1388316664849 0 connected
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master - 0
1388316665850 0 connected
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected 0-5000
```

700070017002
5001100007001

```
127.0.0.1:7001> CLUSTER ADDSLOTS 5001 5002 5003 5004 ... 10000
OK
```

10001163837002

```
127.0.0.1:7002> CLUSTER ADDSLOTS 10001 10002 10003 10004 ... 16383
OK
```

CLUSTER ADDSLOTS16384

```
127.0.0.1:7000> CLUSTER INFO
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:3
```

```
cluster_size:3
cluster_current_epoch:0
cluster_stats_messages_sent:2699
cluster_stats_messages_received:2617
127.0.0.1:7000> CLUSTER NODES
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master - 0
1388317426165 0 connected 10001-16383
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master - 0
1388317427167 0 connected 5001-10000
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected 0-5000
```

集群节点信息

CLUSTER ADDSLOTS

17.2.1 集群节点信息

clusterNode 结构体

```
struct clusterNode {
    // ...
    unsigned char slots[16384/8];
    int numslots;
    // ...
};
```

slots 是一个 bit array，大小为 16384/8=2048 字节，
表示 16384 个槽位。

Redis 0 号节点负责 16383 号槽位，slots 数组大小为 16384 字节，
每个字节表示 8 个槽位，i 表示第 i 个槽位。

· slots 数组中第 i 个槽位的值为 1 表示该槽位被占用。

· 每个 slots 都有一个 i 值，从 0 开始，直到 i

图 17-9 显示了 slots 数组中 0 到 7 的值，以及 1 到 16383 的值。0 到 7 的值是 1，而 1 到 16383 的值是 0。

字节	slots[0]								slots[1] ~ slots[2047]									
索引	0	1	2	3	4	5	6	7	8	9	10	11	12	...	16381	16382	16383	
值	1	1	1	1	1	1	1	1	0	0	0	0	0	...	0	0	0	

图 17-9 每个 slots 的值

图 17-10 显示了 slots 数组中 1 到 3、5 到 8、9 到 10 的值。1 到 3 的值是 0，而 5 到 8、9 到 10 的值是 1。

字节	slots[0]								slots[1]								...	slots[2047]			
索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16382	16383	
值	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0	0	0	0	

图 17-10 每个 slots 的值

每个 slots 都有一个 O 值，从 1 开始，直到 O。slots 数组中 1 到 16383 的值是 0，而 1 到 16383 的值是 1。

每个 numslots 都有一个 slots 值，从 1 开始，直到 numslots。

图 17-9 显示了 slots 数组中 8 到 16383 的值。17-10 显示了 slots 数组中 6 到 16383 的值。

17.2.2 槽位分配策略

集群中每个节点都维护一个clusterNode[]和slots[]和numslots[]
其中clusterNode[]和slots[]和numslots[]分别表示该节点在集群中的位置、
该节点负责的槽位编号、该节点负责的槽位数量。

例如，节点7000、7001、7002在集群中的位置如下：

· 节点7000负责的槽位是7001~7002，其中slots[]数组如下：
槽位编号0~5000，槽位17-11。

· 节点7001负责的槽位是7000~7002，其中slots[]数组如下：
槽位编号5001~10000，槽位17-12。

· 节点7002负责的槽位是7000~7001，其中slots[]数组如下：
槽位编号10001~16383，槽位17-13。

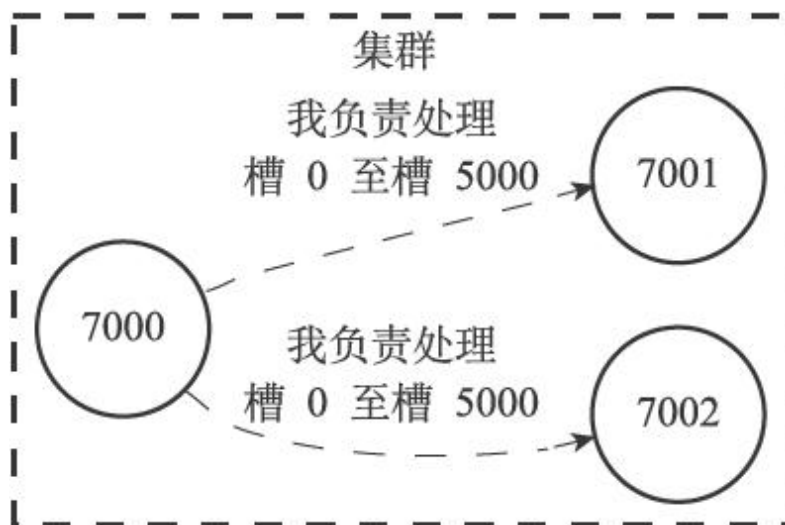


图17-11 7000向7001和7002发送消息

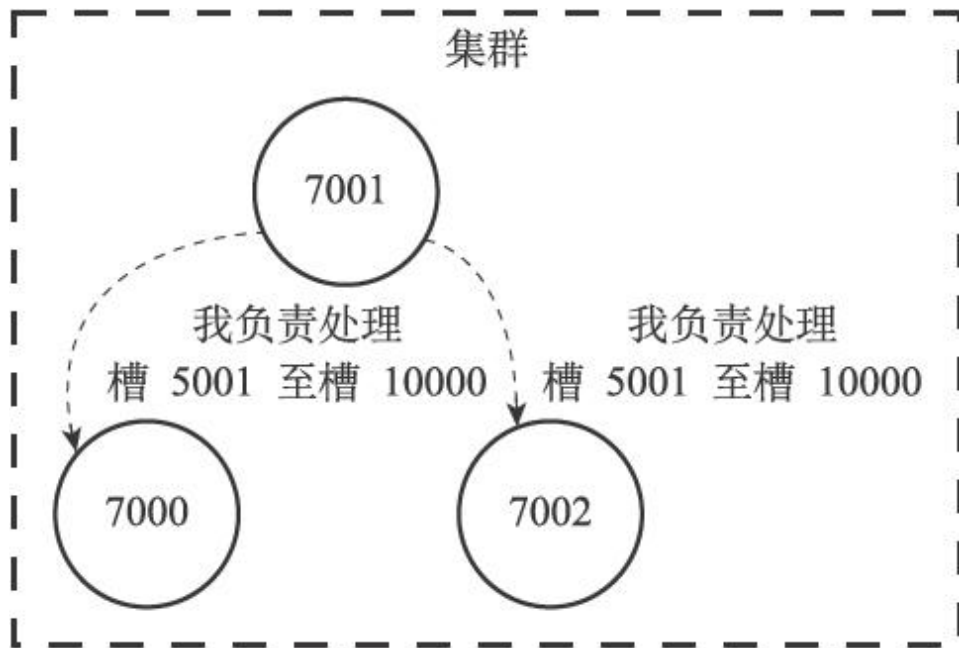


图17-12 7001向7000和7002发送消息

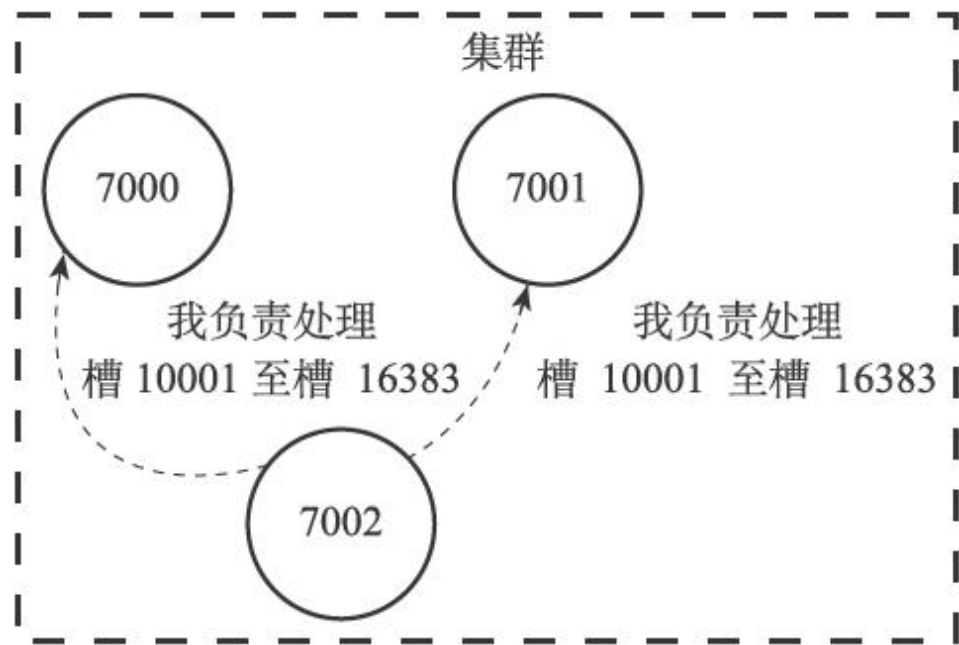


图17-13 7002向7000和7001发送消息

clusterState.nodes[]B[]slots[]A[]
clusterState.nodes[]B[]clusterNode[]slots[]
[]

slots[]
slots[]clusterNode[]
[]16384[]

17.2.3 []

clusterState[]slots[]16384[]

```
typedef struct clusterState {  
    // ...  
    clusterNode *slots[16384];  
    // ...  
} clusterState;
```

slots[]16384[]clusterNode[]
·[]slots[i][]NULL[]i[]
·[]slots[i][]clusterNode[]i[]
clusterNode[]

[]7000[]7001[]7002[]clusterState[]
slots[]17-14[]

· slots[0] slots[5000] 7000 clusterNode
0 5000 7000

· slots[5001] slots[10000] 7001
clusterNode 5001 10000 7001

· slots[10001] slots[16383] 7002
clusterNode 10001 16383 7002

clusterNode.slots
clusterState.slots

· clusterNode.slots i
clusterState.nodes
clusterNode.slots i
O N N clusterState.nodes clusterNode

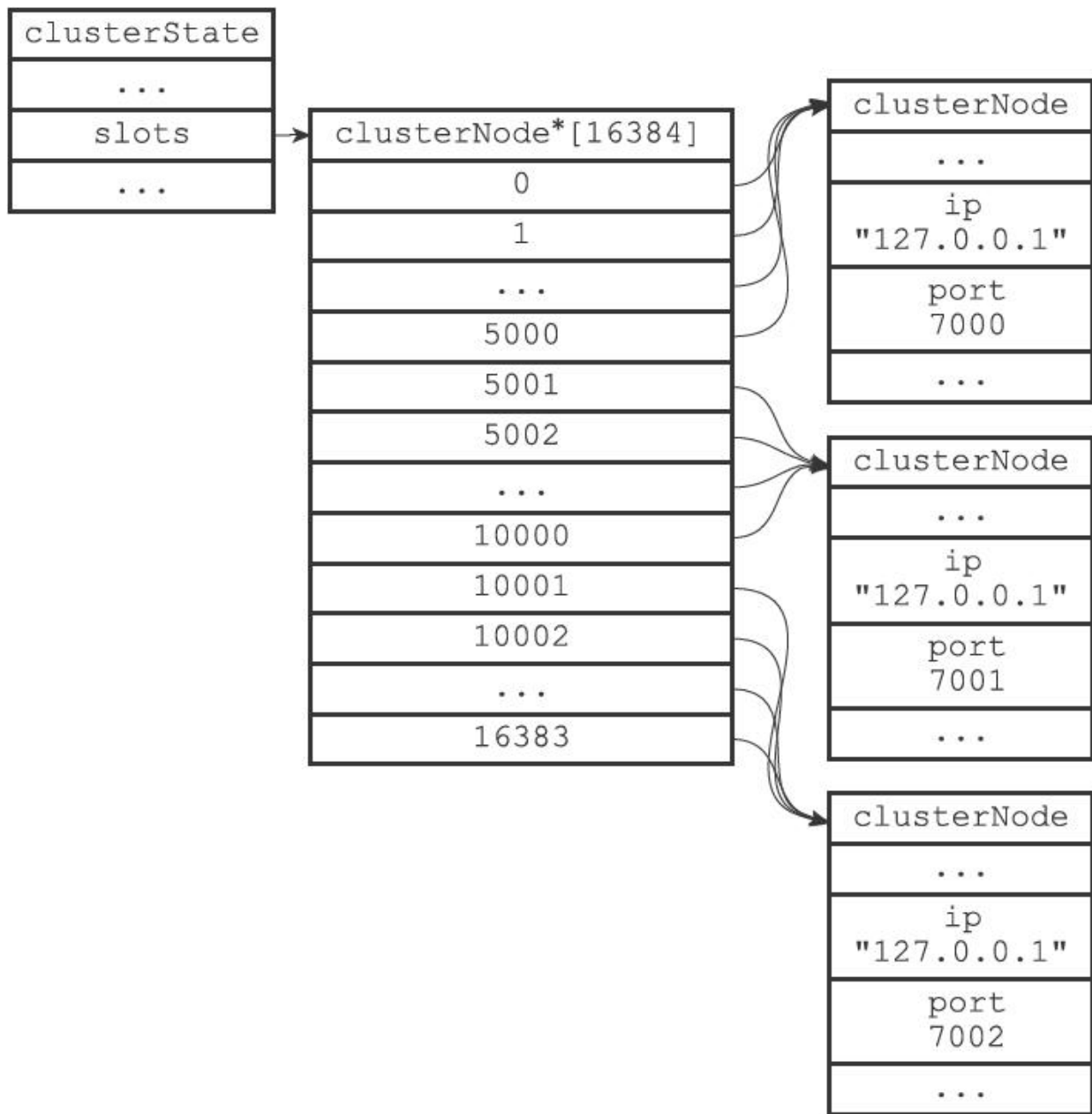


图17-14 clusterState中的slots

· 遍历 `clusterState.slots` 数组，找到每个 `i` 对应的 `clusterState.slots[i]` 的值，然后遍历 `clusterState.slots[i]` 的值，找到每个 `O` 的值。

图17-14展示了slots数组的索引10002指向的节点信息。
 图17-15展示了如何访问slots[10002]的值。

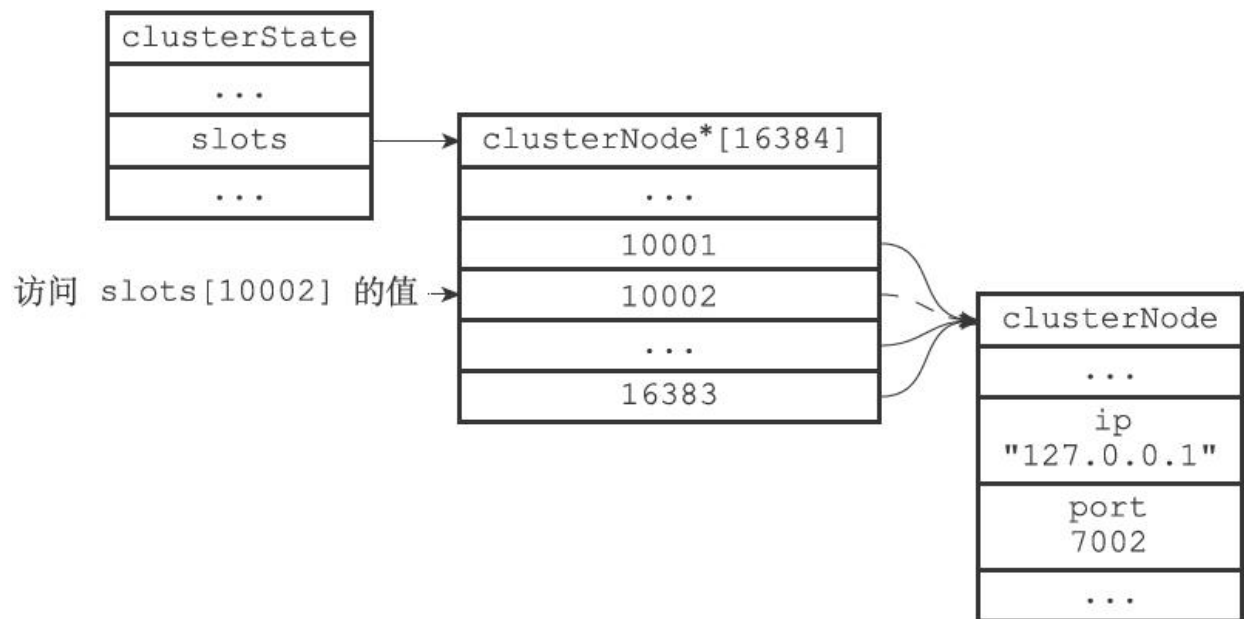


图17-15 访问slots[10002]的值

图17-16展示了如何访问clusterState.slots数组中的元素。

图17-17展示了如何访问clusterNode.slots数组中的元素。

图17-18展示了如何访问Redis集群中的clusterNode.slots数组。

```
clusterState.slots[A] = clusterNode.slots
clusterNode.slots =
```

```
clusterState.slots[clusterNode.slots]
clusterNode.slots =
```

17.2.4 CLUSTER ADDSLOTS

CLUSTER ADDSLOTS

```
CLUSTER ADDSLOTS <slot> [slot ...]
```

CLUSTER ADDSLOTS

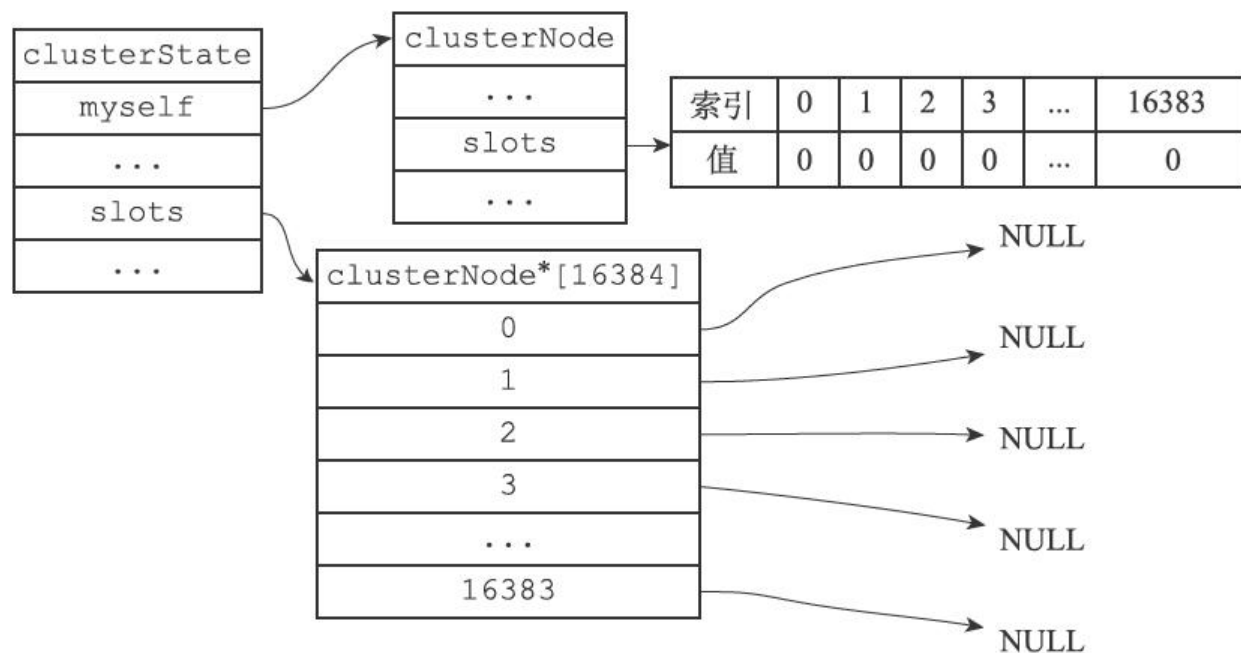
```
def CLUSTER_ADDSLOTS(*all_input_slots):
    #
    for i in all_input_slots:
        #
        #
        if clusterState.slots[i] != NULL:
            reply_error()
            return
        #
        #
        for i in all_input_slots:
            #
            clusterState
            slots
```

```

    #
    slots[i]
    clusterNode
    clusterState.slots[i] = clusterState.myself
    #
    clusterNode
    slots
    #
    i
    1
    setSlotBit(clusterState.myself.slots, i)

```

17-16 clusterState clusterState.slots
 NULL clusterNode.slots 0



17-16 clusterState

17-16

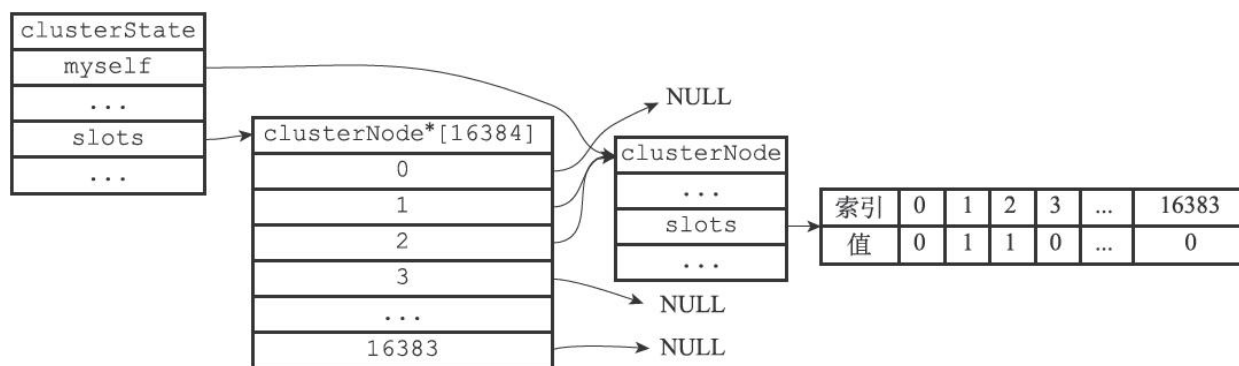
CLUSTER ADDSLOTS 1 2

1 2 clusterState 17-17

·clusterState.slots 1 2

clusterNode

·clusterNode.slots 1 2 1



17-17 CLUSTER ADDSLOTS clusterState

CLUSTER ADDSLOTS

17.3 一致性哈希

一致性哈希算法由 Ben 伯曼在 1997 年提出，其核心思想是：将键和节点都映射到哈希空间（一个圆环）上，通过计算键和节点的哈希值，确定键应该存储在哪个节点上。

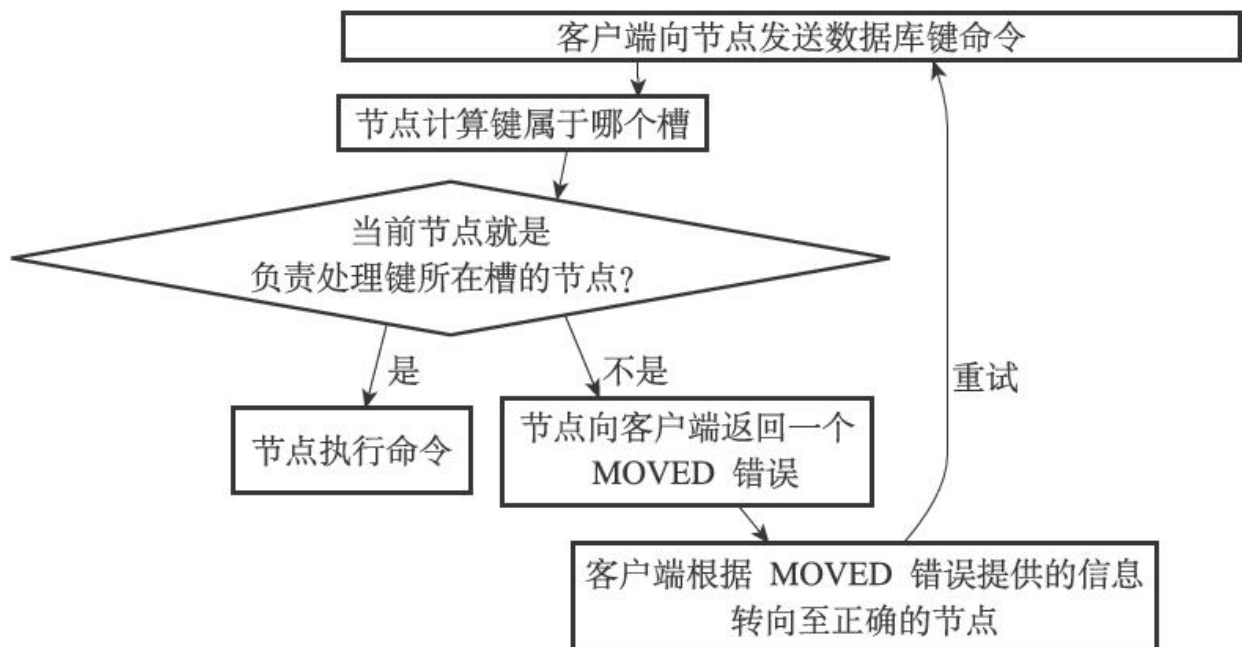
一致性哈希算法具有以下特点：

- 节点加入或离开时，只有部分键需要迁移，不会影响整个系统的可用性。

- 节点加入或离开时，系统会自动重新计算键的哈希值，并将键迁移到新的节点上，这个过程称为 **MOVED** 操作。

一致性哈希算法的实现通常依赖于一个哈希表，该表将键映射到节点上。当节点加入或离开时，系统会根据哈希表的值来重新分配键。

图 17-18 一致性哈希算法流程图



17-18 实验实验实验实验

实验实验实验实验实验7000实验7001实验7002实验实验实验实验实验实验
实验7000实验实验实验实验实验实验实验7000实验

```
127.0.0.1:7000> SET date "2013-12-31"  
OK
```

实验date实验2022实验实验7000实验实验

实验实验实验实验实验实验实验实验实验7001实验实验实验

```
127.0.0.1:7000> SET msg "happy new year!"  
-> Redirected to slot [6257] located at 127.0.0.1:7001  
OK  
127.0.0.1:7001> GET msg  
"happy new year!"
```

实验实验msg实验6257实验7001实验实验实验实验实验实验实验
7000实验实验

·实验实验实验实验7000实验SET实验实验实验7000实验实验实验MOVED
实验实验实验实验实验7001实验

·实验实验实验实验7001实验实验实验实验7001实验SET实验实验实验实验
7001实验实验

MOVED
Redis

17.3.1

key

```
def slot_number(key):  
    return CRC16(key) & 16383
```

CRC16keykeyCRC-16&16383
016383key

CLUSTER KEYSLOT<key>

```
127.0.0.1:7000> CLUSTER KEYSLOT "date"  
(integer) 2022  
127.0.0.1:7000> CLUSTER KEYSLOT "msg"  
(integer) 6257  
127.0.0.1:7000> CLUSTER KEYSLOT "name"  
(integer) 5798  
127.0.0.1:7000> CLUSTER KEYSLOT "fruits"  
(integer) 14943
```

CLUSTER KEYSLOT

```
def CLUSTER_KEYSLLOT(key):  
    #  
    
```

```
slot = slot_number(key)
#
reply_client(slot)
```

17.3.2 集群槽位管理

集群槽位管理是指对集群槽位进行管理，包括槽位的分配、回收、迁移等。在 Redis 中，槽位管理是通过 `clusterState.slots` 来管理的。

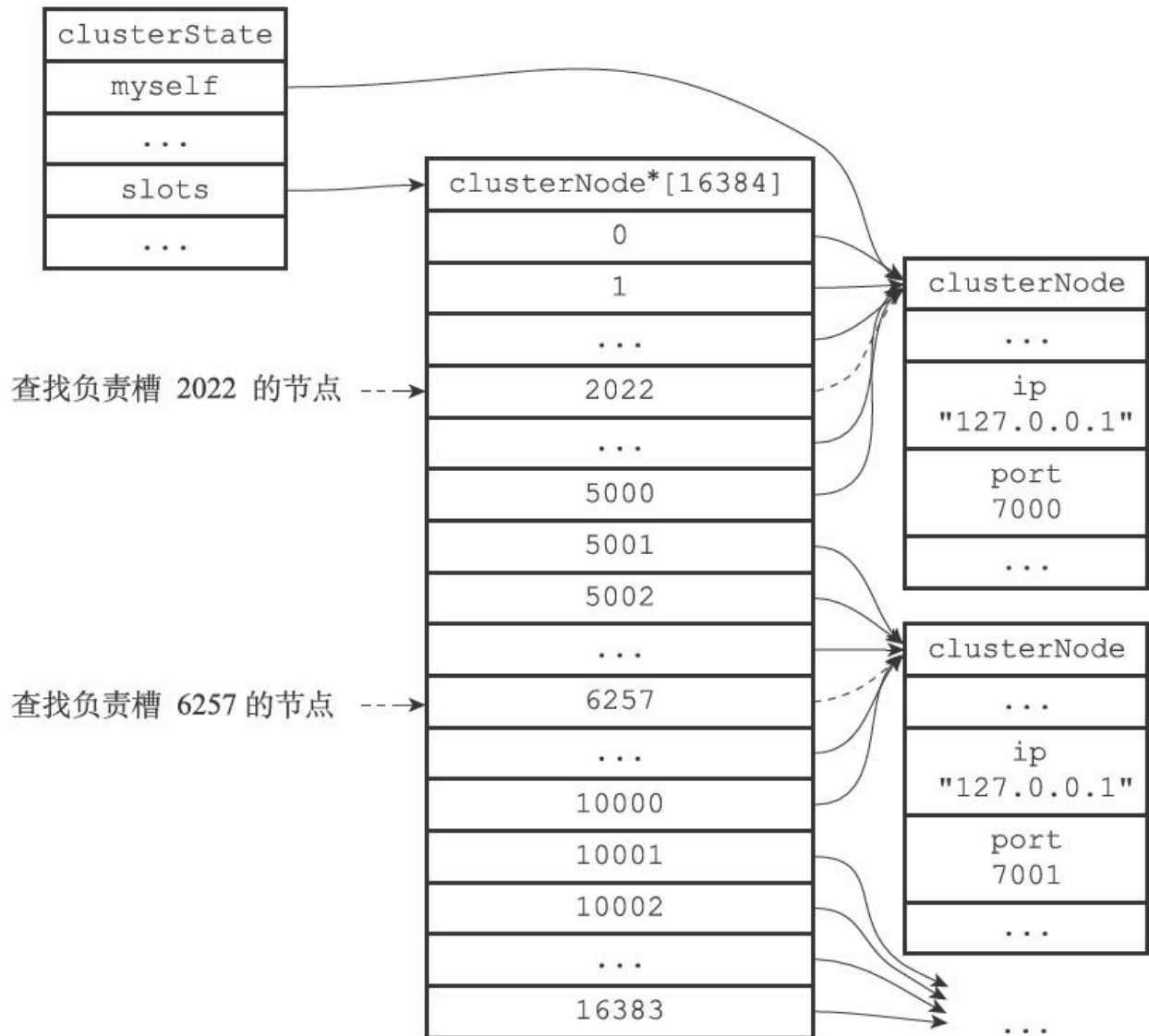
1. 槽位分配：当集群中的节点加入时，需要为其分配槽位。可以通过 `clusterState.slots[i]` 来访问槽位 `i` 的信息。

2. 槽位回收：当集群中的节点离开时，需要回收其占用的槽位。可以通过 `clusterState.slots[i]` 来访问槽位 `i` 的信息，并设置 `clusterNode` 为 `MOVED`，表示该槽位已被回收。

在 Redis 中，槽位管理是通过 `clusterState.slots` 来管理的。在 Redis 17-19 版本中，槽位数量是 7000。在 Redis 2022 版本中，槽位数量增加到了 16384。可以通过 `SET date "2013-12-31"` 来设置槽位 `date` 的过期时间。在 Redis 2022 版本中，可以通过 `clusterState.slots[2022]` 来访问槽位 `2022` 的信息。在 Redis 2022 版本中，可以通过 `clusterState.myself` 来访问当前节点的信息。在 Redis 2022 版本中，可以通过 `SET msg "happy new year"` 来设置槽位 `msg` 的过期时间。在 Redis 2022 版本中，可以通过 `clusterState.slots[6257]` 来访问槽位 `6257` 的信息。

在 Redis 2022 版本中，可以通过 `SET msg "happy new year"` 来设置槽位 `msg` 的过期时间。在 Redis 2022 版本中，可以通过 `clusterState.slots[6257]` 来访问槽位 `6257` 的信息。

```
clusterState.myself[6257]
7000 clusterState.slots[6257] clusterNode
IP 127.0.0.1 7001 MOVED 6257
127.0.0.1:7001 6257 7001
```



```
17-19  7000 clusterState
```

17.3.3 MOVED□□

MOVED
MOVED

MOVED

MOVED <slot> <ip>:<port>

slot ip port slot IP

MOVED 10086 127.0.0.1:7002

10086 IP 127.0.0.1 7002

MOVED 789 127.0.0.1:7000

789 IP 127.0.0.1 7000

MOVED MOVED IP
slot 7000
7001

127.0.0.1:7000> SET msg "happy new year!"
-> Redirected to slot [6257] located at 127.0.0.1:7001
OK
127.0.0.1:7001>

图17-20 客户端向节点7000发送SET消息并收到MOVED响应

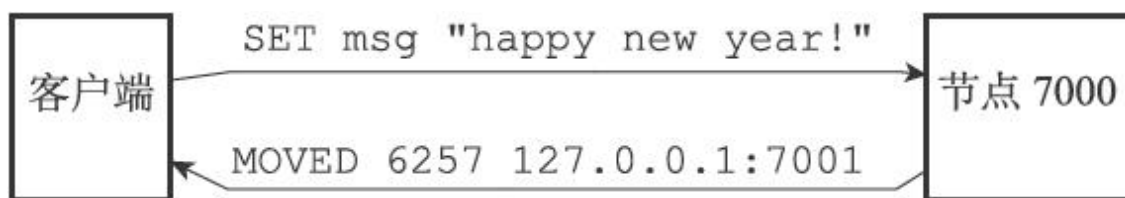


图17-20 客户端向节点7000发送SET消息并收到MOVED响应

图17-21 客户端向节点7001发送SET消息并收到OK响应

图

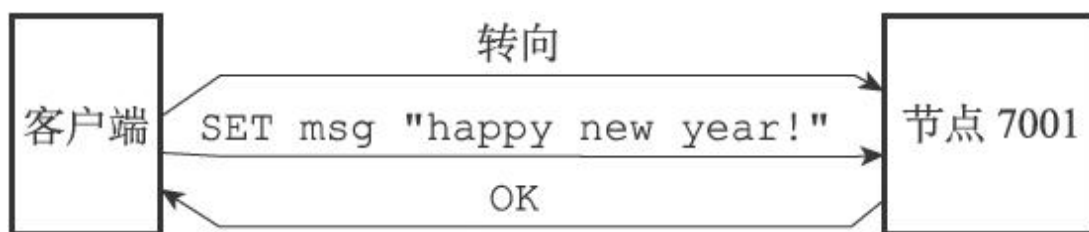


图17-21 客户端向节点7001发送SET消息并收到OK响应

当客户端收到MOVED响应时，它应该知道消息已经被移动到另一个节点上。

图

当客户端收到MOVED响应时，它应该知道消息已经被移动到另一个节点上。

图

客户端收到MOVED响应

```
redis-cliMOVEDMOVED
MOVEDMOVED

```

```
$ redis-cli -c -p 7000 #
127.0.0.1:7000> SET msg "happy new year!"
-> Redirected to slot [6257] located at 127.0.0.1:7001
OK
127.0.0.1:7001>
```

```
stand aloneredis-cli7000
MOVED
```

```
$ redis-cli -p 7000 #
127.0.0.1:7000> SET msg "happy new year!"
(error) MOVED 6257 127.0.0.1:7001
127.0.0.1:7000>
```

```
redis-cliMOVED
MOVED
```

17.3.4

9Redis

0 Redis

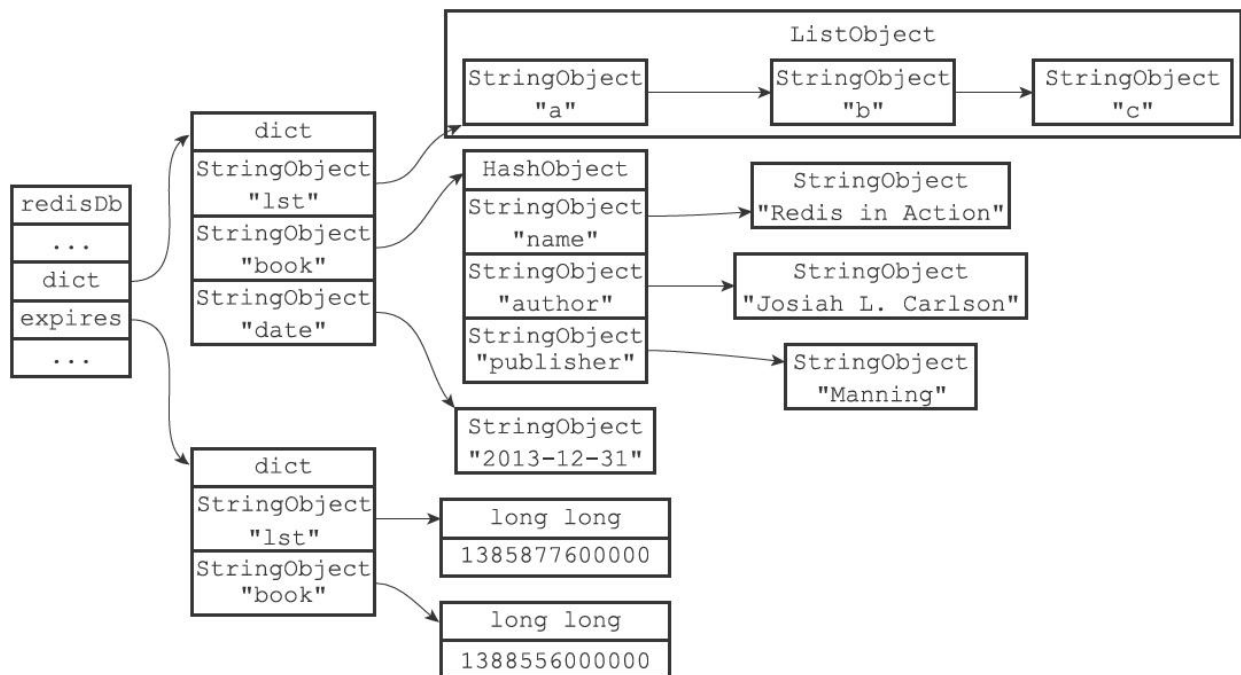
17-22 7000 "lst"

"book" "date" "lst" "book"

clusterState

slots_to_keys

```
typedef struct clusterState {
    // ...
    zskiplist *slots_to_keys;
    // ...
} clusterState;
```



17-22 7000

slots_to_keys[]score[]
member[]

·
slots_to_keys[]

·slots_to_keys[]
[]

17-22700017-23
slots_to_keys[]

·"book"1337.0"book"1337

·"date"2022.0"date"2022

·"lst"3347.0"lst"3347

slots_to_keys[]
CLUSTER GETKEYSINSLOT<slot>
<count>countslot
slots_to_keys[]

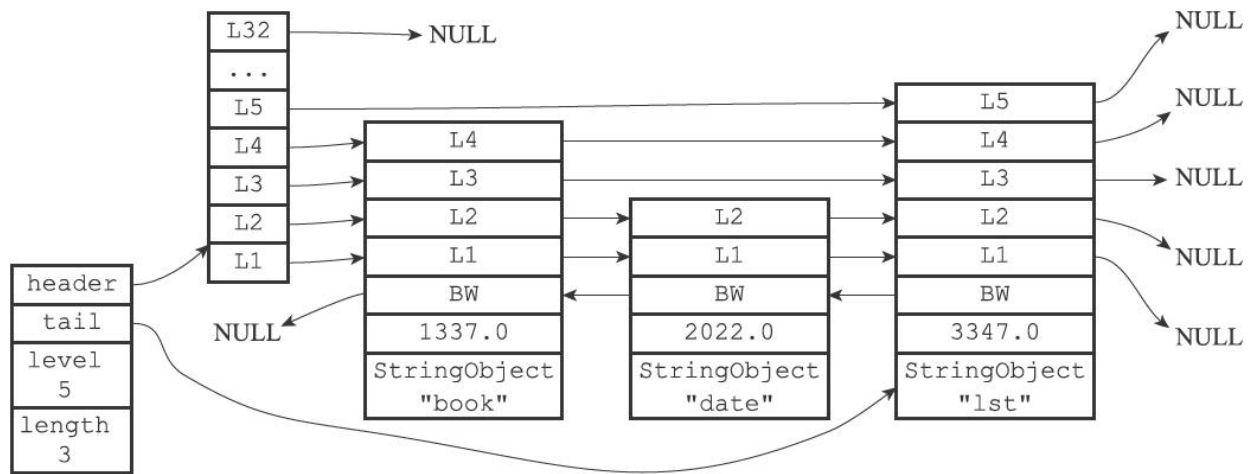


图17-23 7000 slots_to_keys

17.4 Redis

Redis是一个开源的分布式数据库，它支持多种数据类型，如字符串、列表、集合、有序集合、哈希表、位图等。Redis还支持主从复制、哨兵、集群等功能。

Redis的集群模式支持高可用和可扩展性。在集群模式下，Redis实例可以分布在不同的服务器上，并且可以自动进行故障转移和负载均衡。

Redis集群的每个节点都有一个唯一的ID，并且每个节点都负责一部分数据。Redis集群的节点可以通过IP地址和端口号进行访问。

```
$ redis-cli -c -p 7000
127.0.0.1:7000> CLUSTER MEET 127.0.0.1 7003
OK
127.0.0.1:7000> cluster nodes
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master - 0 0 0
connected 0-5000
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master - 0
1388635782831 0 connected 5001-10000
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master - 0
1388635782831 0 connected 10001-16383
04579925484ce537d3410d7ce97bd2e260c459a2 127.0.0.1:7003 master - 0
1388635782330 0 connected
```

Redis集群的每个节点都有一个唯一的ID，并且每个节点都负责一部分数据。Redis集群的节点可以通过IP地址和端口号进行访问。

Redis集群的每个节点都有一个唯一的ID，并且每个节点都负责一部分数据。

```
127.0.0.1:7000> cluster nodes
51549e625cfda318ad27423a31e7476fe3cd2939 :0 myself,master -0 0 0
connected 0-5000
68eef66df23420a5862208ef5b1a7005b806f2ff 127.0.0.1:7001 master -0
1388635782831 0 connected 5001-10000
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26 127.0.0.1:7002 master -0
1388635782831 0 connected 10001-15000
04579925484ce537d3410d7ce97bd2e260c459a2 127.0.0.1:7003 master -0
1388635782330 0 connected 15001-16383
```

redis 命令

Redis 集群的 redis-trib 命令 Redis 集群的 redis-trib 命令

redis-trib 命令 slot 命令

1 redis-trib 命令 CLUSTER
SETSLOT <slot> IMPORTING <source_id> 命令
import 命令 slot 命令

2 redis-trib 命令 CLUSTER
SETSLOT <slot> MIGRATING <target_id> 命令 slot
migrate 命令

3 redis-trib 命令 CLUSTER GETKEYSINSLOT <slot>
<count> 命令 count 命令 slot 命令 key name

4 redis-trib

MIGRATE<target_ip><target_port><key_name>0<timeout>

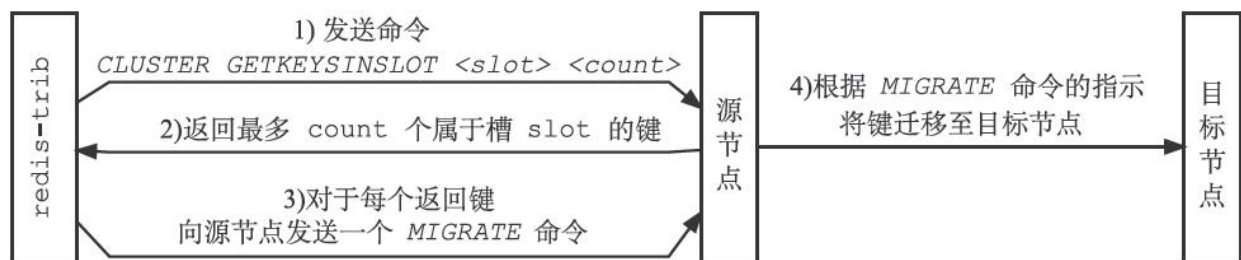
5 redis-trib 3 4 slot

17-24

6 redis-trib CLUSTER

SETSLOT<slot>NODE<target_id> slot

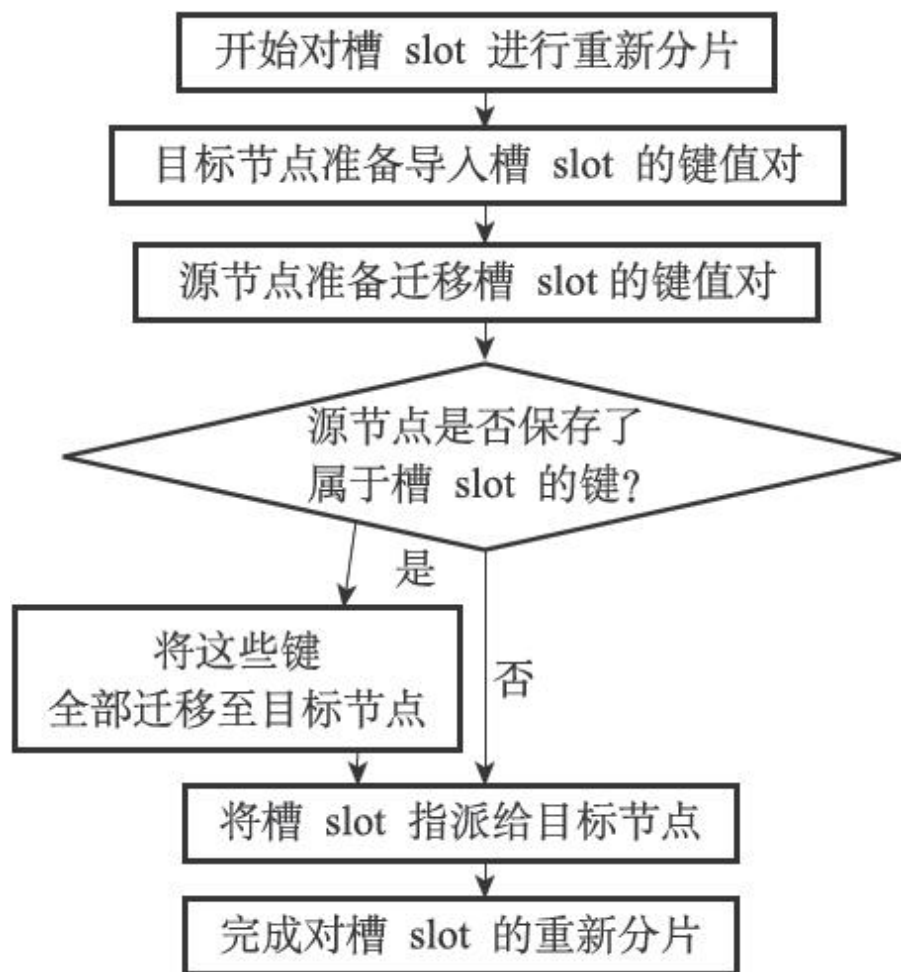
slot



17-24

17-25 slot

redis-trib



□17-25 □□slot□□□□□□□□

17.5 ASK

当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令，源节点返回 ASK 错误。

当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令，源节点返回 ASK 错误。

· 当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令，源节点返回 ASK 错误。

· 当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令，源节点返回 ASK 错误。

当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令，源节点返回 ASK 错误。

图 17-26 当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令

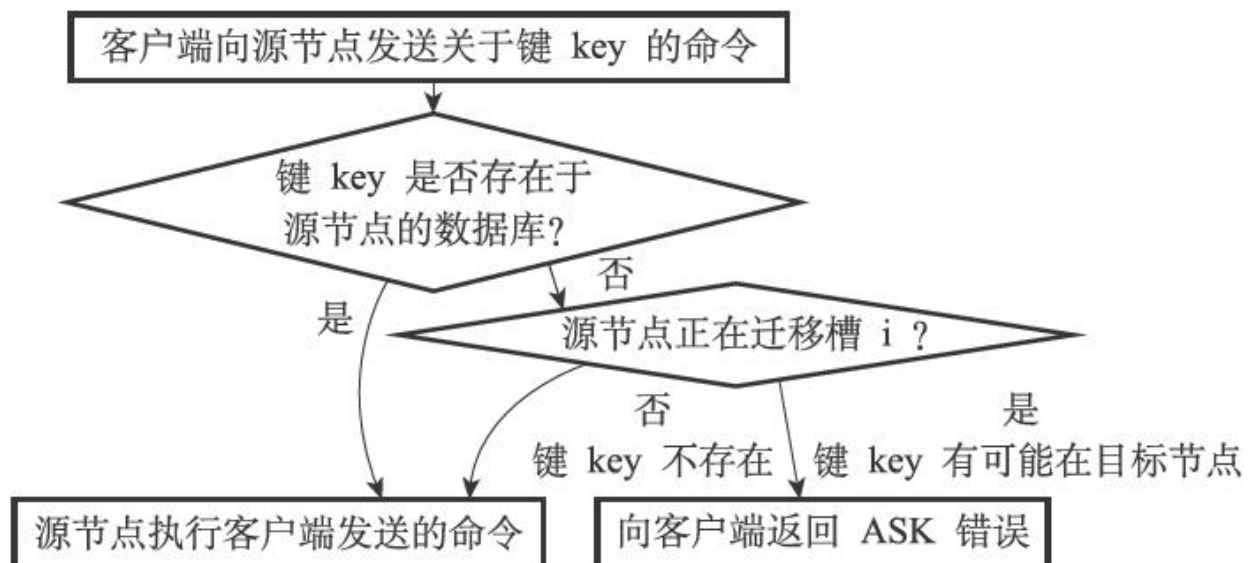


图 17-26 当源节点正在迁移槽 i 时，客户端向源节点发送关于键 key 的命令

redis-cli -p 7002 --c 7003 --s 16198 --t "is" --l "love" --o
redis-cli -p 7002 --c 7003 --s 16198 --t "is" --l "love" --o

redis-cli -p 7002 --c 7003 --s 16198 --t "is" --l "love" --o

```
127.0.0.1:7002> GET "is"
"you get the key 'is'"
```

redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o
redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o

```
127.0.0.1:7002> GET "love"
-> Redirected to slot [16198] located at 127.0.0.1:7003
"you get the key 'love'"
127.0.0.1:7003>
```

redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o

redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o
redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o
redis-cli -p 7002 --c 7003 --s 16198 --t "love" --l "love" --o

```
$ redis-cli -p 7002
127.0.0.1:7002> GET "love"
(error) ASK 16198 127.0.0.1:7003
```



```
redis-cli ASK
MOVED
redis-cli ASK
```

```
ASK
MOVED
```

17.5.1 CLUSTER SETSLOT IMPORTING

```
clusterState.importing_slots_from
```

```
typedef struct clusterState {
    // ...
    clusterNode *importing_slots_from[16384];
    // ...
} clusterState;
```

```
importing_slots_from[i] NULL clusterNode
clusterNode[i]
```

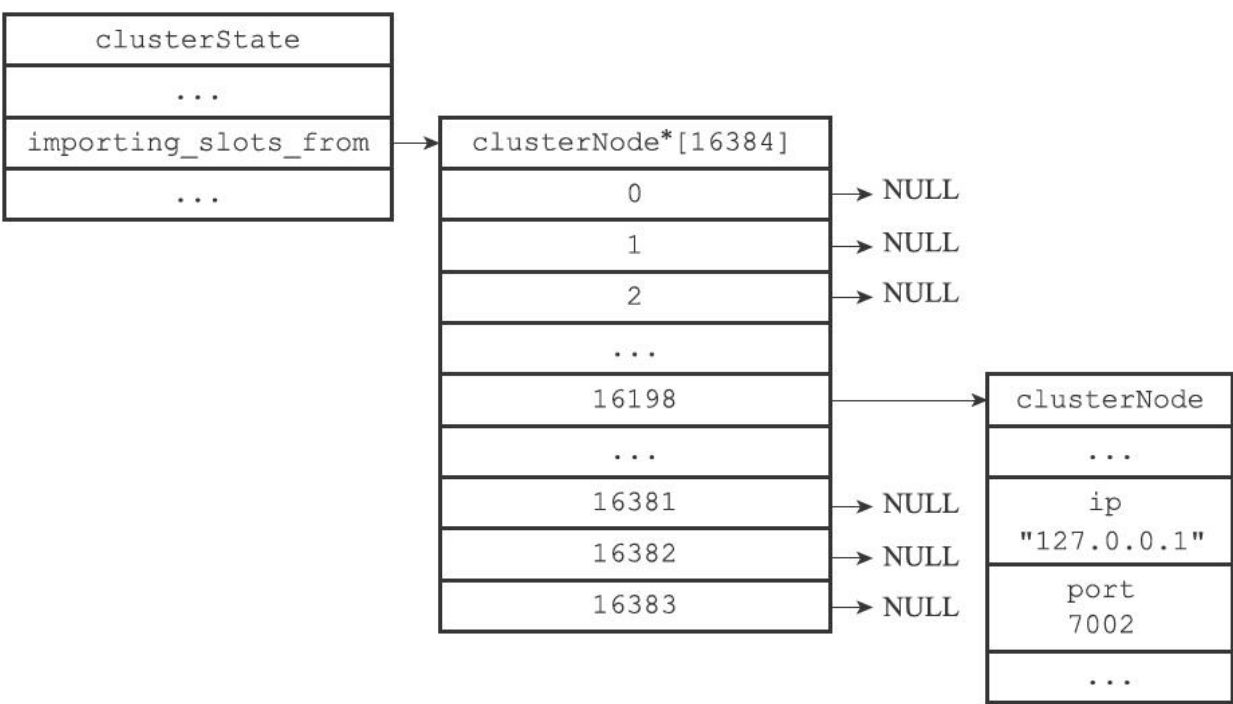
```
CLUSTER SETSLOT <i> IMPORTING <source_id>
```

clusterState.importing_slots_from[i]
source_idclusterNode

7003

```
# 9dfb...  
7002  
ID  
127.0.0.1:7003> CLUSTER SETSLOT 16198 IMPORTING  
9dfb4c4e016e627d9769e4c9bb0d4fa208e65c26  
OK
```

7003clusterState.importing_slots_from17-
27



17-27 7003importing_slots_from

17.5.2 CLUSTER SETSLOT MIGRATING

`clusterState.migrating_slots_to` is an array of pointers to `clusterNode` objects.

For

```
typedef struct clusterState {  
    // ...  
    clusterNode *migrating_slots_to[16384];  
    // ...  
} clusterState;
```

`migrating_slots_to[i]` is a pointer to a `clusterNode` object or `NULL` if no node is migrating to slot `i`.

The command is used to migrate a slot to a target node.

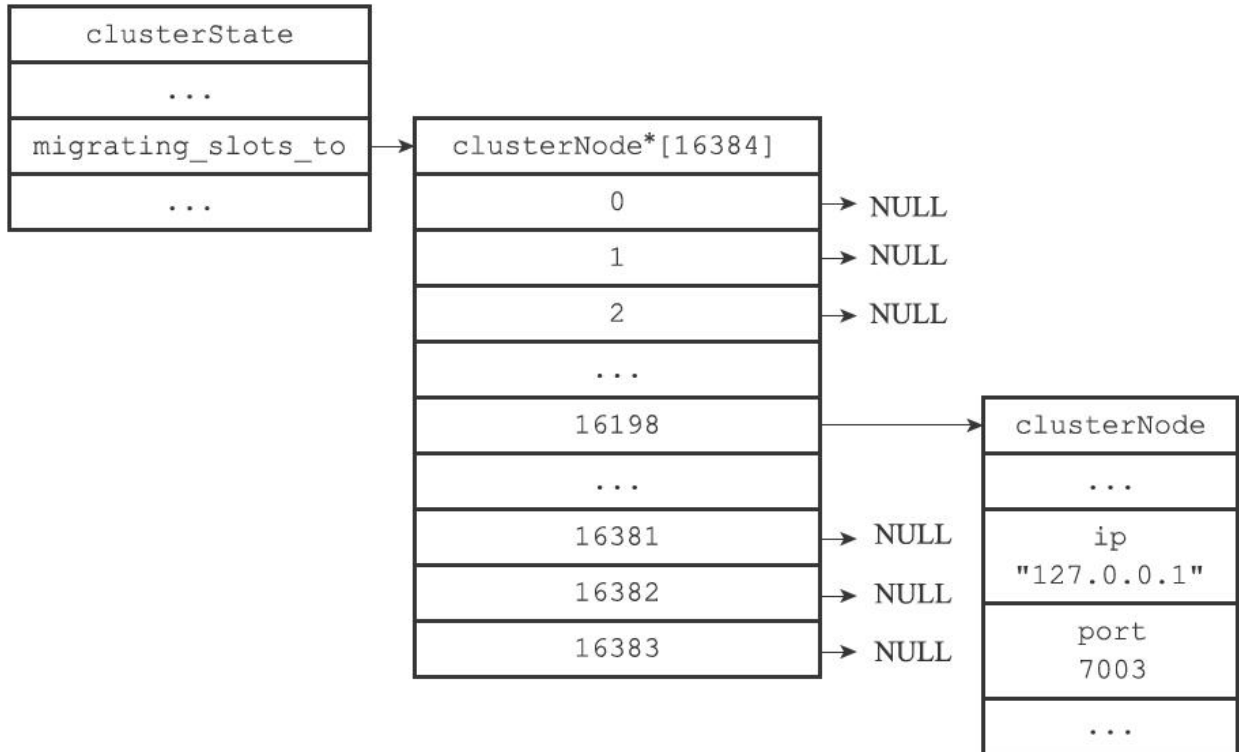
```
CLUSTER SETSLOT <i> MIGRATING <target_id>
```

The command migrates slot `i` to the node identified by `target_id`. The command returns the `clusterNode` object of the target node.

The command returns the ID of the target node.

```
# 0457...  
7003  
ID  
127.0.0.1:7002> CLUSTER SETSLOT 16198 MIGRATING  
04579925484ce537d3410d7ce97bd2e260c459a2  
OK
```

7002 clusterState.migrating_slots_to 17-28



17-28 7002 migrating_slots_to

17.5.3 ASK

key key i
key

key
clusterState.migrating_slots_to[i] key i

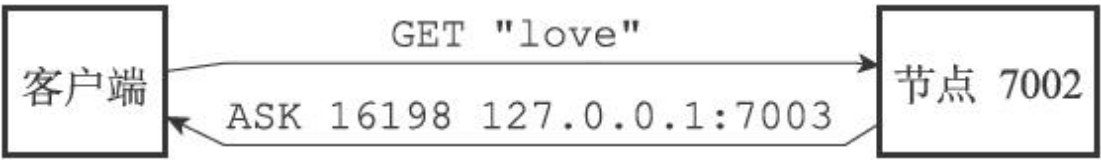
客户端向节点7002发送ASK消息，询问key
 节点7002返回16198槽位

GET
 "love"

节点7002返回16198槽位，并返回"love"
 客户端收到16198槽位，并返回clusterState.migrating_slots_to[16198]
 7002槽位，并返回16198槽位

ASK 16198 127.0.0.1:7003

客户端向节点7003发送ASK消息，询问16198槽位
 节点7003返回17-29槽位



17-29 槽位，并返回7002槽位ASK
 客户端收到ASK消息，并返回IP
 客户端收到ASKING消息

客户端向节点7002发送消息

ASK 16198 127.0.0.1:7003

节点7003向客户端发送消息

ASKING

客户端

GET "love"

节点

"you get the key 'love'"

节点17-30



节点17-30 节点7003

17.5.4 ASKING

ASKING 和 REDIS_ASKING

```
def ASKING():  
    #  
    client.flags |= REDIS_ASKING  
    #  
    OK  
    reply("OK")
```

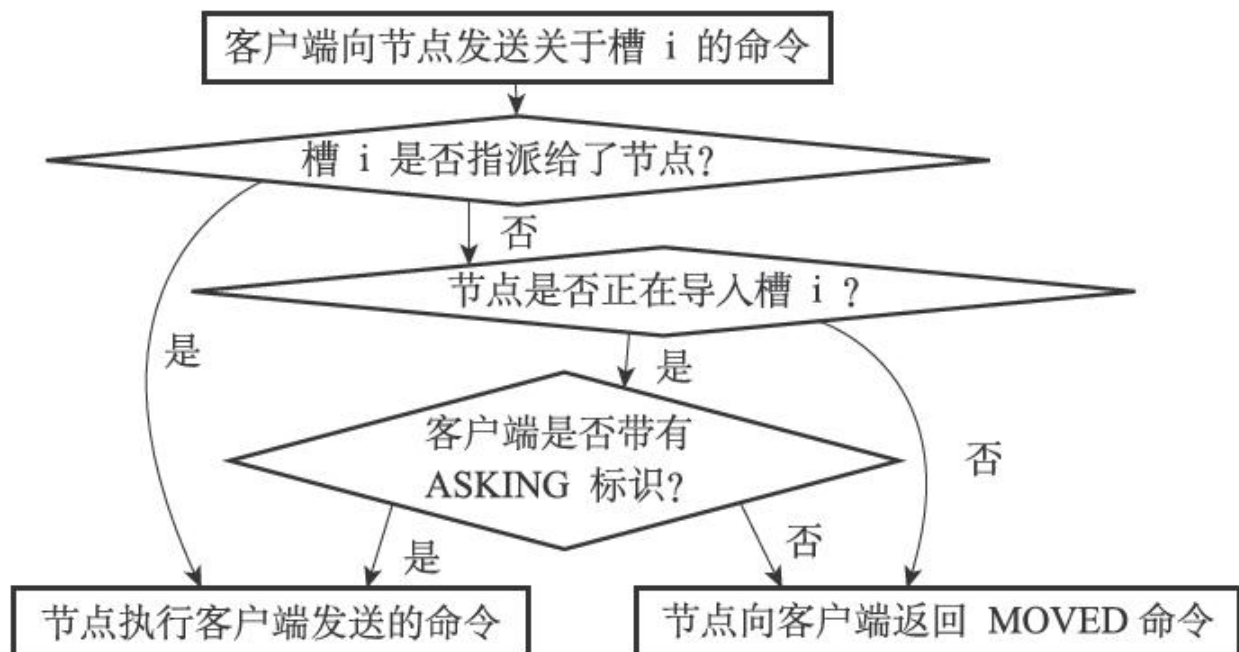
槽 i 是否指派给了节点？

MOVED

clusterState.importing_slots_from[i]

REDIS_ASKING

17-31



□17-31 □□□□□□□□□□□□□□□□

[illegible]

```
redis-cli 16198 7003
```

```
$ ./redis-cli -p 7003
127.0.0.1:7003> GET "love"
(error) MOVED 16198 127.0.0.1:7002
```

7003 16198 16198 7002
7003 MOVED 7002

```

GET / HTTP/1.1
Host: 10.0.0.1
User-Agent: Mozilla/5.0 (Windows NT 6.0; rv:2.0) Gecko/20100101 Firefox/4.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Cache-Control: max-age=0
Connection: close
Cookie: PHPSESSID=7003

```

```
127.0.0.1:7003> ASKING
OK
127.0.0.1:7003> GET "love"
"you get the key 'love'"
```

[illegible]

GET 7003 GET
GET REDIS Asking

```
127.0.0.1:7003> ASK #
REDIS Asking
OK
127.0.0.1:7003> GET "love" #
REDIS Asking
"you get the key 'love'"
127.0.0.1:7003> GET "love" # REDIS Asking
(error) MOVED 16198 127.0.0.1:7002
```

17.5.5 ASK MOVED

ASK MOVED

· MOVED
MOVED
MOVED

· ASK
ASK
ASK
ASK

17.6 Redis 集群

Redis 集群由多个 master 和 slave 组成，每个 master 和 slave 都是一个 Redis 实例，它们通过 TCP 连接组成一个集群。

集群由 7000、7001、7002、7003、7004、7005 组成，其中 7000 是主节点，17-32 是 slave 节点。

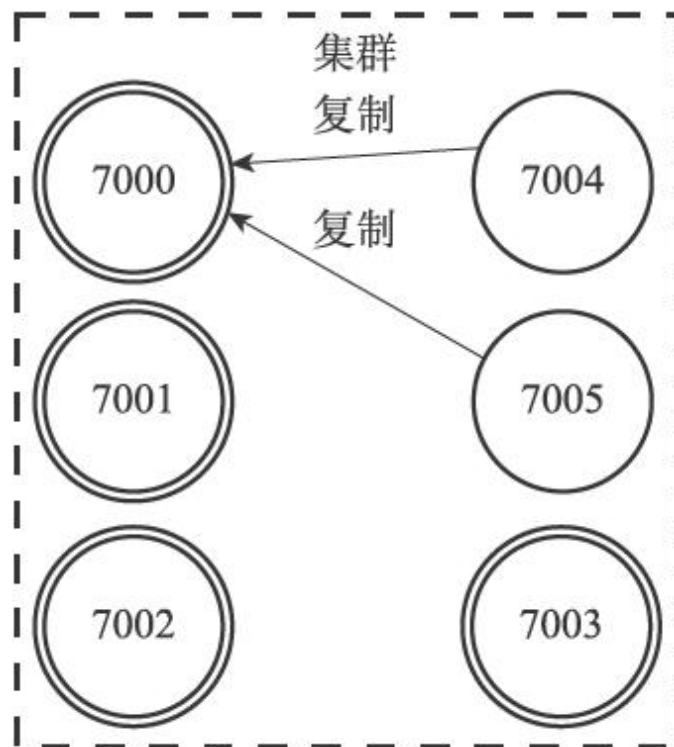


图 17-32 Redis 集群 7004 和 7005 复制 7000

图 17-1 Redis 集群的组成

图17-1 节点角色和状态

节点	角色	状态	工作
7000	主节点	在线	负责处理槽 0 至槽 5000
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	从节点	在线	复制节点 7000
7005	从节点	在线	复制节点 7000

节点7000是主节点，负责处理槽0至槽5000。节点7001、7002、7003也是主节点，分别负责处理槽5001至槽10000、10001至槽15000、15001至槽16383。节点7004和7005是从节点，分别复制节点7000的数据。

节点7004和7005是主节点，负责处理槽7004至槽7005。节点7000是主节点，负责处理槽0至槽5000。节点7005是从节点，复制节点7000的数据。图17-33展示了节点角色和状态。

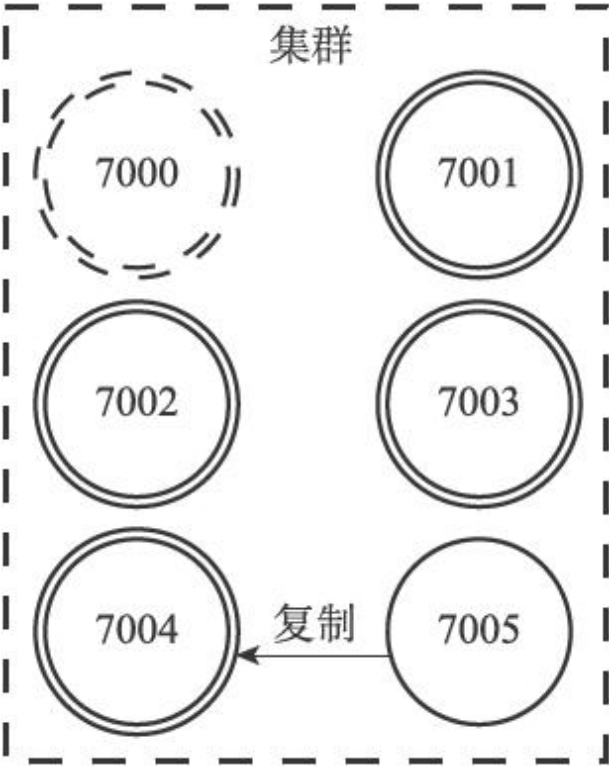


图17-33 节点7004的副本

图17-2展示了节点7000的副本分布情况，节点7000的副本分布在节点7001、7002、7003、7004和7005上。

图17-2 副本分布情况

节点	角色	状态	工作
7000	主节点	下线	负责处理槽 0 至槽 5000（因为故障转移已经完成，所以该工作已经无效。）
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	主节点	在线	负责处理槽 0 至槽 5000
7005	从节点	在线	复制节点 7004

图17-2展示了节点7000的副本分布情况，节点7000的副本分布在节点7001、7002、7003、7004和7005上。

图17-34

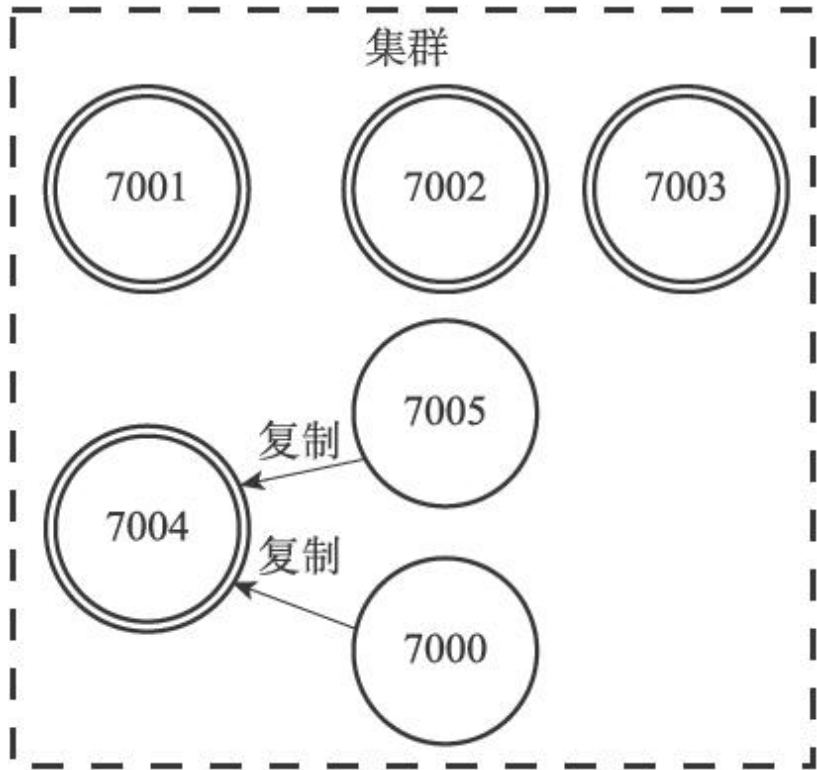


图17-34 节点7000复制节点7004

图17-3 节点7000复制节点7004

图17-3 节点7000复制节点7004

节点	角色	状态	工作
7000	从节点	在线	复制节点 7004
7001	主节点	在线	负责处理槽 5001 至槽 10000
7002	主节点	在线	负责处理槽 10001 至槽 15000
7003	主节点	在线	负责处理槽 15001 至槽 16383
7004	主节点	在线	负责处理槽 0 至槽 5000
7005	从节点	在线	复制节点 7004

节点7000复制节点7004

图

17.6.1 复制

复制命令

```
CLUSTER REPLICATE <node_id>
```

复制命令的格式为 `CLUSTER REPLICATE node_id`

· 复制命令的格式为 `CLUSTER REPLICATE clusterState.nodes[node_id]`
其中 `clusterNode` 是 `clusterState.myself.slaveof` 指向的节点
信息。

```
struct clusterNode {  
    // ...  
    //  
    struct clusterNode *slaveof;  
    // ...  
};
```

· 复制命令的格式为 `CLUSTER REPLICATE clusterState.myself.flags`
`REDIS_NODE_MASTER` 和 `REDIS_NODE_SLAVE` 是节点标志
信息。

· 复制命令的格式为 `CLUSTER REPLICATE clusterState.myself.slaveof`
`clusterNode` 是 `IP` 地址和 `Redis` 节点
信息。 `SLAVEOF` 是

图17-35 端口7004的clusterState

·clusterState.myself.flags为REDIS_NODE_SLAVE

7004

·clusterState.myself.slaveof为7000

7000

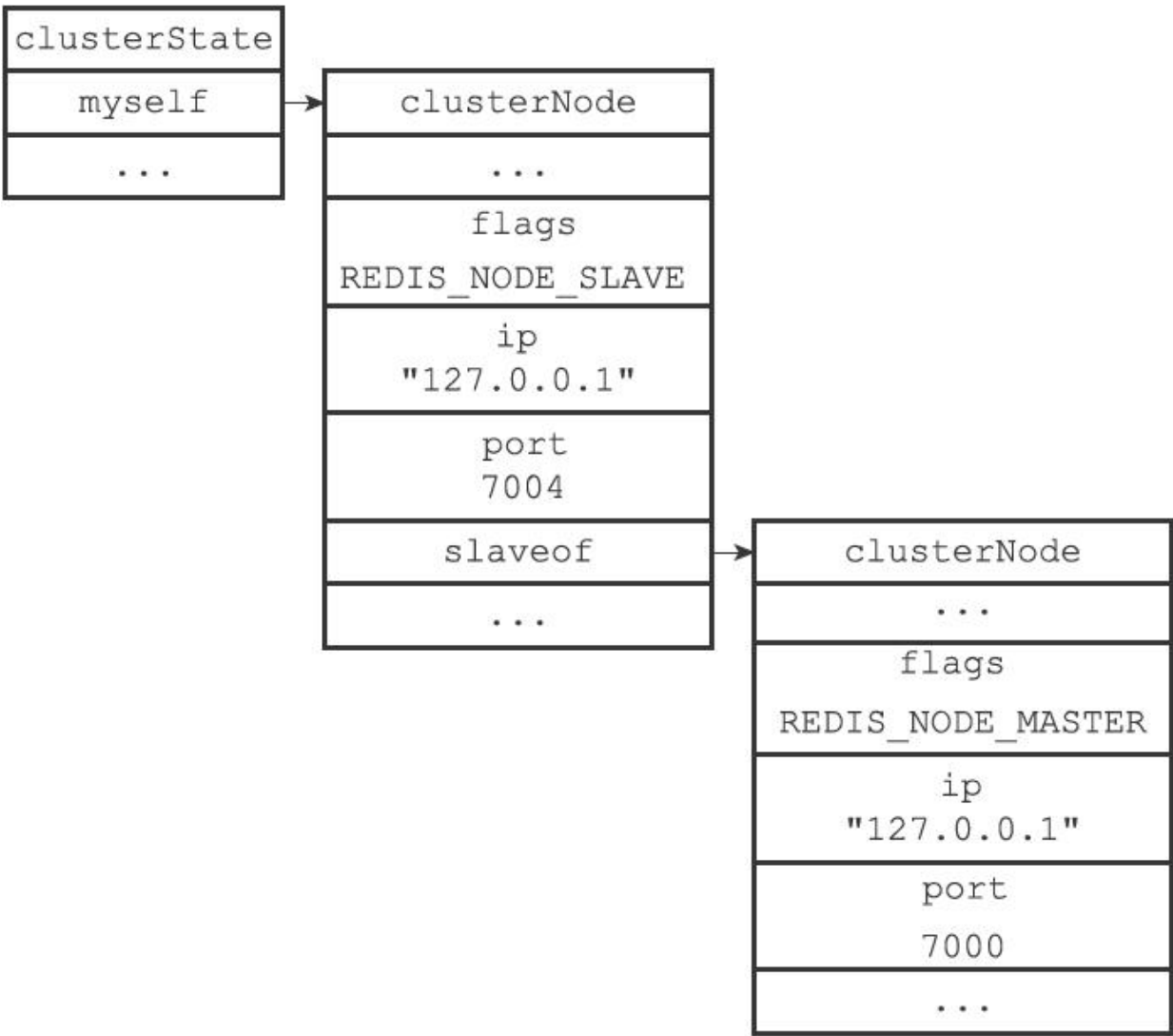


图17-35 端口7004的clusterState

1. 在代码中，我们定义了一个名为 `clusterNode` 的结构体，用于存储集群节点的信息。

2. 在代码中，我们定义了一个名为 `clusterNode` 的结构体，用于存储集群节点的信息。

```

struct clusterNode {
    // ...
    //
    int numslaves;
    //
    //
    //
    struct clusterNode **slaves;
    // ...
};
    
```

3. 在代码中，我们定义了一个名为 `clusterNode` 的结构体，用于存储集群节点的信息。

4. 在代码中，我们定义了一个名为 `clusterNode` 的结构体，用于存储集群节点的信息。

5. 在代码中，我们定义了一个名为 `clusterNode` 的结构体，用于存储集群节点的信息。

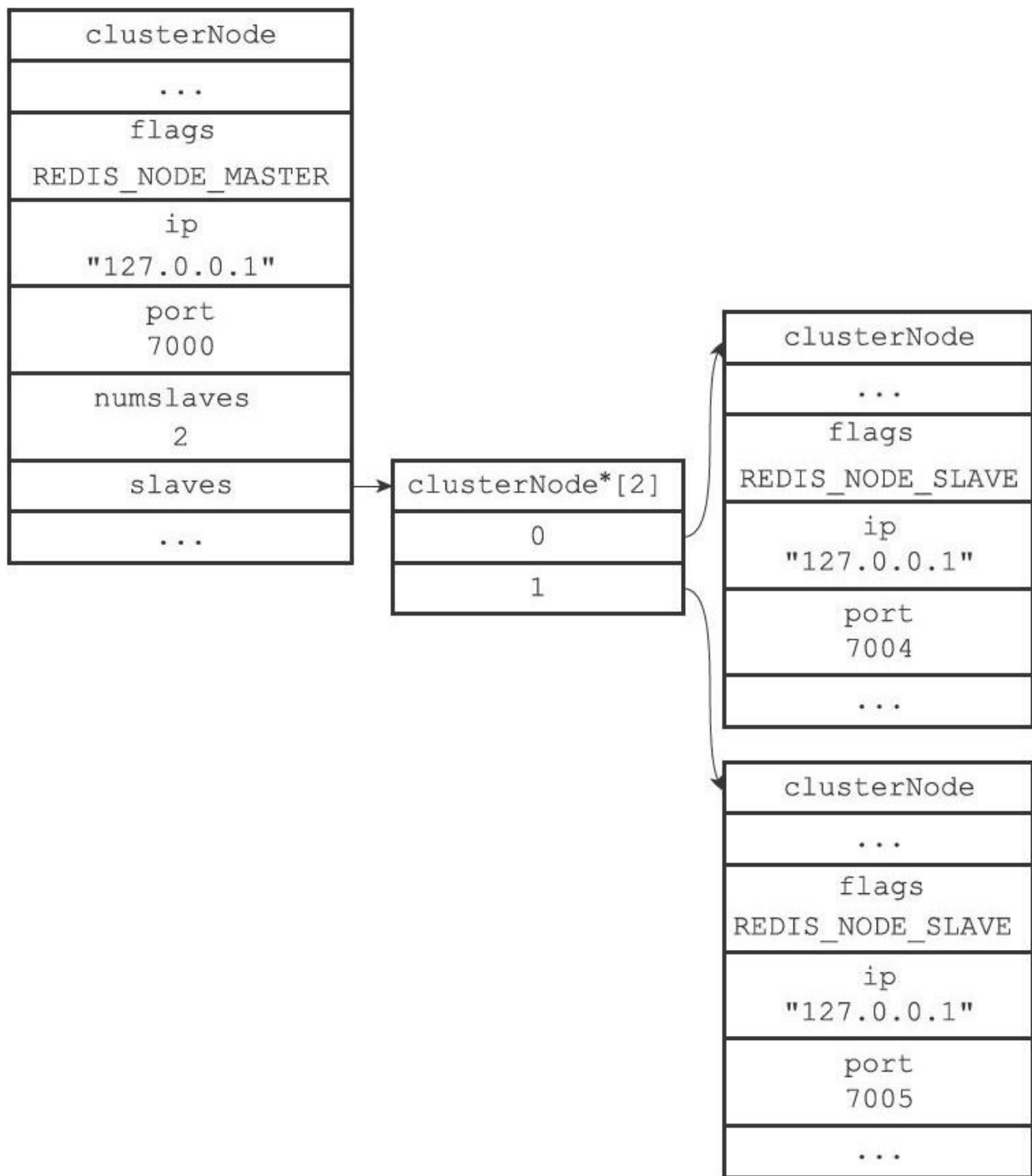


图17-36 主节点（端口7000）的clusterNode结构

17.6.2 主节点

redis 7001 ping 7000
7001 ping 7000 pong 7001 ping 7000
7001 ping 7000 probable fail PFAIL

redis 7001 ping 7000
7001 pong 7001
clusterState.nodes[7000] clusterNode flags
REDIS_NODE_PFAIL 7000
17-37

clusterNode
...
flags
REDIS_NODE_MASTER & REDIS_NODE_PFAIL
ip
"127.0.0.1"
port
7000
...

图 17-37 redis 7000 clusterNode

redis 7001 ping 7000
7001 PFAIL 7001 FAIL

```

    clusterState.nodes[C] == clusterNode[B]
    failure report[C] == clusterNode[B].fail_reports

```

```
struct clusterNode {  
    // ...  
    //  
    [0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]  
    list *fail_reports;  
    // ...  
};
```

clusterNodeFailReport

```
struct clusterNodeFailReport {
    //
    struct clusterNode *node;
    //
    mstime_t time;
} typedef clusterNodeFailReport;
```

□□□□□□□□7001□□□□□□7002□□□□7003□□□□□□□□□□
7002□□□□7003□□□□□□7000□□□□□□□□□□□□7001□□□□7000
□□□17-38□□□□□□□□

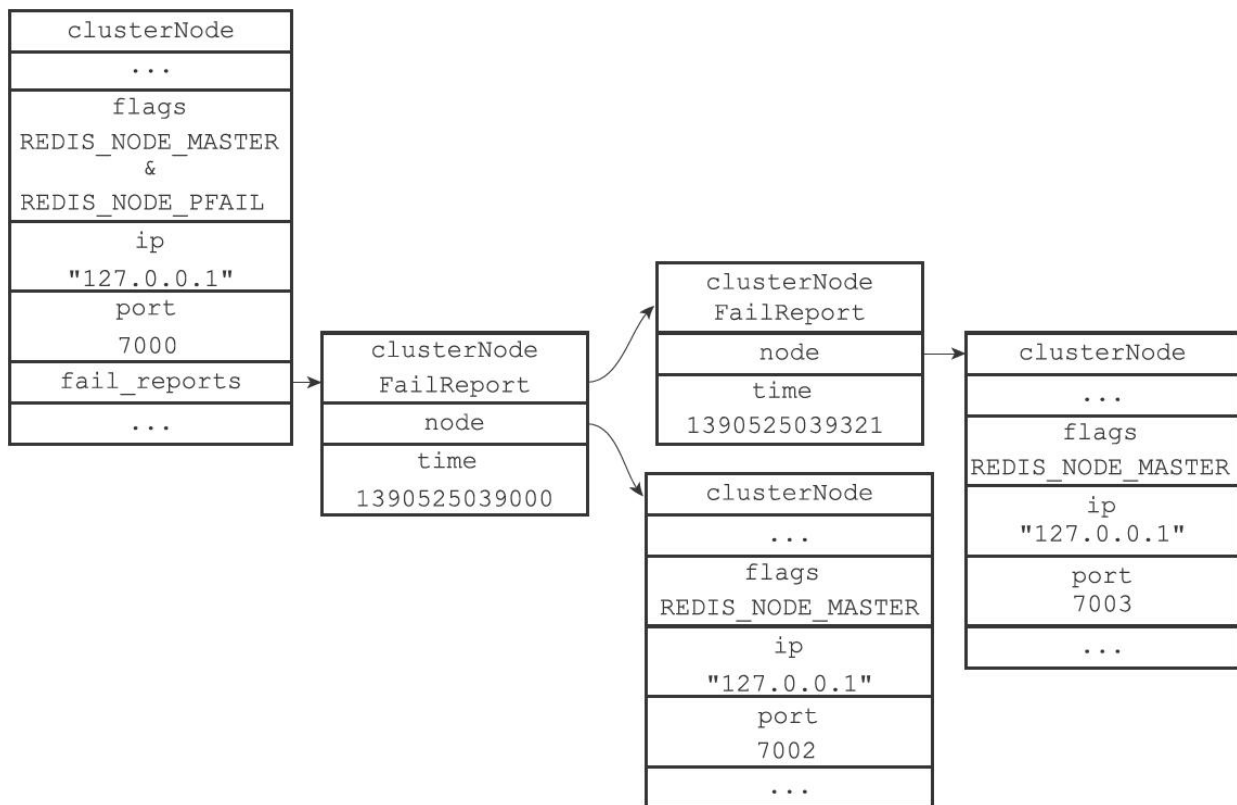


图17-38 节点7000的状态变化

节点7000在收到节点7001的失败报告后，会将节点7001的状态从REDIS_NODE_MASTER变为REDIS_NODE_PFAIL，并将节点7001的失败报告添加到失败报告中。节点7000在收到节点7001的失败报告后，会将节点7001的状态从REDIS_NODE_MASTER变为REDIS_NODE_PFAIL，并将节点7001的失败报告添加到失败报告中。

图17-38展示了节点7002和节点7003的状态变化。节点7002在收到节点7001的失败报告后，会将节点7001的状态从REDIS_NODE_MASTER变为REDIS_NODE_PFAIL，并将节点7001的失败报告添加到失败报告中。节点7002在收到节点7001的失败报告后，会将节点7001的状态从REDIS_NODE_MASTER变为REDIS_NODE_PFAIL，并将节点7001的失败报告添加到失败报告中。

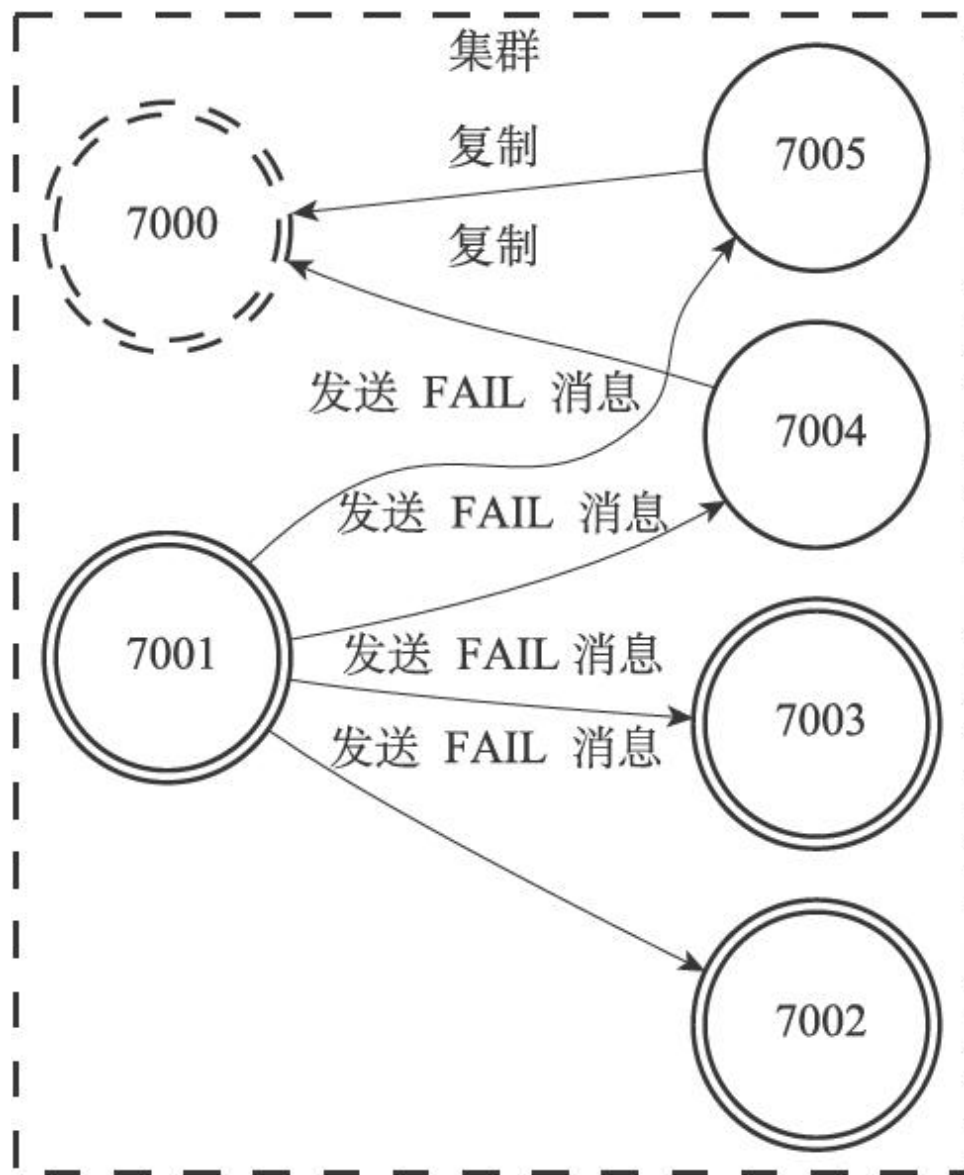


图17-39 节点7001发送FAIL消息

17.6.3 故障处理

当集群中某个节点发生故障时，集群中的其他节点会检测到该故障，并启动故障处理流程。故障处理流程包括故障检测、故障定位、故障恢复和故障预防等步骤。

1. 故障检测

2 主节点向所有从节点发送 SLAVEOF no one

3 主节点向所有从节点发送 PONG

4 主节点向所有从节点发送 PONG

主节点向所有从节点发送 PONG

5 主节点向所有从节点发送 PONG

17.6.4 主节点向从节点发送 PONG

主节点向从节点发送 PONG

主节点向从节点发送 PONG

1 主节点向从节点发送 PONG

2 主节点向从节点发送 PONG

3 主节点向从节点发送 PONG

主节点向从节点发送 PONG

4 主节点向从节点发送 PONG

CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST

主节点向从节点发送 PONG

5 收到消息的节点收到消息后，如果消息中的 `CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK` 消息中的 `term` 大于自己的 `term`，则更新自己的 `term` 为消息中的 `term`。

6 收到消息的节点收到消息后，如果消息中的 `CLUSTERMSG_TYPE_FAILOVER_AUTH_ACK` 消息中的 `term` 等于自己的 `term`，且消息中的 `leader` 不等于自己的 `leader`，则更新自己的 `leader` 为消息中的 `leader`。

7 收到消息的节点收到消息后，如果消息中的 `term` 大于自己的 `term`，且消息中的 `term` 大于 $N/2 + 1$ ，则更新自己的 `term` 为消息中的 `term`。

8 收到消息的节点收到消息后，如果消息中的 `term` 等于自己的 `term`，且消息中的 `term` 大于 $N/2 + 1$ ，则更新自己的 `term` 为消息中的 `term`。

9 收到消息的节点收到消息后，如果消息中的 `term` 等于自己的 `term`，且消息中的 `term` 大于 $N/2 + 1$ ，则更新自己的 `term` 为消息中的 `term`。

在 Redis 中，哨兵（Sentinel）用于监控主节点的健康状况，并在主节点故障时进行 leader election。Raft 是一种分布式共识算法，用于在分布式系统中达成一致。

17.7 消息

Redis 的消息传递模型是简单的，它使用 `message` 消息传递模型。消息由 `sender` 发送，由 `receiver` 接收。图 17-40 展示了消息传递模型。

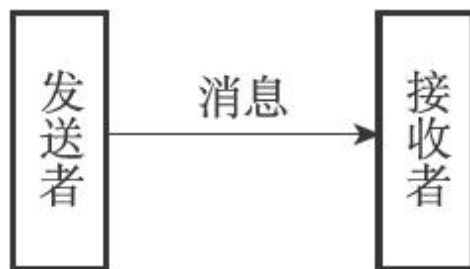


图 17-40 消息传递模型

Redis 的消息传递模型如下：

- `MEET`：用于集群成员之间的消息传递。集群成员通过 `MEET` 消息来通知其他成员它们已经加入集群。

- `PING`：用于集群成员之间的消息传递。集群成员通过 `PING` 消息来通知其他成员它们是否仍然存活。集群成员 A 通过 `PING` 消息来通知集群成员 B。集群成员 B 通过 `PONG` 消息来响应集群成员 A。集群成员 A 通过 `PING` 消息来通知集群成员 B。集群成员 B 通过 `PONG` 消息来响应集群成员 A。


```

uint16_t type;
//
uint16_t count;
//
uint16_t MEET;
uint16_t PING;
uint16_t PONG;
uint16_t Gossip;
uint16_t count;
//
uint64_t currentEpoch;
//
uint64_t configEpoch;
//
uint64_t ID;
char sender[REDIS_CLUSTER_NAMELEN];
//
unsigned char myslots[REDIS_CLUSTER_SLOTS/8];
//
uint64_t REDIS_NODE_NULL_NAME;
//
uint40
uint0
char slaveof[REDIS_CLUSTER_NAMELEN];
//
uint16_t port;
//
uint16_t flags;
//
unsigned char state;
//
union clusterMsgData data;
} clusterMsg;

```


myslots
flags

17.7.2 MEET PING PONG

Redis Gossip Gossip
MEET PING PONG
cluster.h/clusterMsgDataGossip

```
union clusterMsgData {  
    // ...  
    // MEET  
    PING  
    PONG  
    struct {  
        //  
        MEET  
        PING  
        PONG  
        // clusterMsgDataGossip  
        clusterMsgDataGossip gossip[1];  
    } ping;  
    //  
};
```

enum {MEET, PING, PONG} enum_t; type_t
enum {MEET, PING, PONG} enum_t

enum {MEET, PING, PONG} enum_t
enum {MEET, PING, PONG} enum_t
clusterMsgDataGossip enum_t

clusterMsgDataGossip enum_t
enum {PING, PONG} enum_t IP enum_t

```
typedef struct {  
    //  
    enum_t  
    char nodename[REDIS_CLUSTER_NAMELEN];  
    //  
    enum_t PING  
    enum_t  
    uint32_t ping_sent;  
    //  
    enum_t PONG  
    enum_t  
    uint32_t pong_received;  
    //  
    enum_t IP  
    enum_t  
    char ip[16];  
    //  
    enum_t  
    uint16_t port;  
    //  
    enum_t  
    uint16_t flags;  
} clusterMsgDataGossip;
```

集群中每个节点都发送 MEET、PING、PONG 消息给其他节点。
clusterMsgDataGossip 消息中包含 clusterMsgDataGossip 消息。
集群中每个节点都发送 MEET、PING、PONG 消息给其他节点。

· 集群中每个节点都发送 MEET、PING、PONG 消息给其他节点。
集群中每个节点都发送 IP 消息给其他节点。

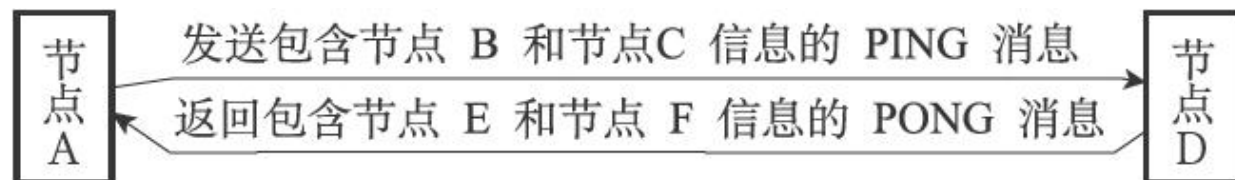
· 集群中每个节点都发送 MEET、PING、PONG 消息给其他节点。
集群中每个节点都发送 clusterMsgDataGossip 消息给其他节点。
clusterNode 消息。

集群中每个节点都发送 PING 消息给其他节点。A、B、C、D、E、F 节点。
集群中每个节点都发送 PING 消息给其他节点。

· 节点 A 发送 D 节点 PING 消息。节点 B 发送 C 节点 D 节点。
PING 消息。节点 B 发送 C 节点。

· 节点 D 发送 A 节点 PONG 消息。节点 E 发送 F 节点 A 节点。
PONG 消息。节点 E 发送 F 节点。

集群中每个节点都发送 17-41 消息。



17-41 PING-PONG

17.7.3 FAIL□□□□

```

00000000A0000B00000000FAIL000000A000000000000000B0
FAIL000000000000FAIL000000000000B0000000

```

```

Gossip
Gossip
FAIL

```

```

FAILcluster.h/clusterMsgDataFail
nodename

```

```
typedef struct {
    char nodename[REDIS_CLUSTER_NAMELEN];
} clusterMsgDataFail;
```

[illegible]

□□□□□□□7000□7001□7002□7003□□□□□□□□□□

```
·0000700100007000000000007001000070020000
700300FAIL0000FAIL0000000000000000700000000000
```


7000

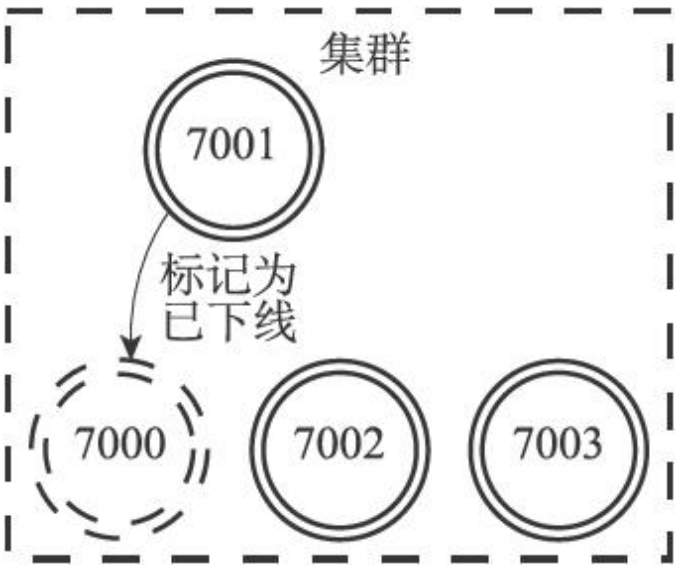
· 7002 7003 7001 FAIL

7000

· 7000

7000

17-42 17-44 FAIL



17-42 7001 7000

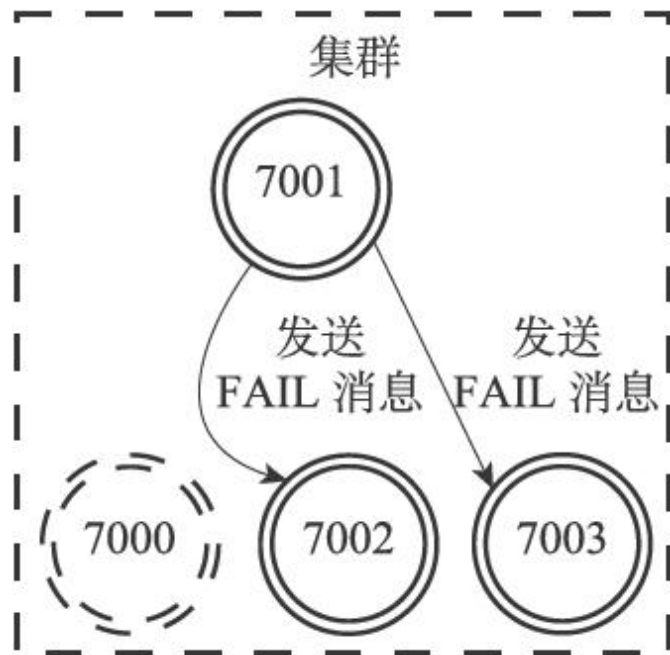


图17-43 节点7001向节点7002和7003发送FAIL消息

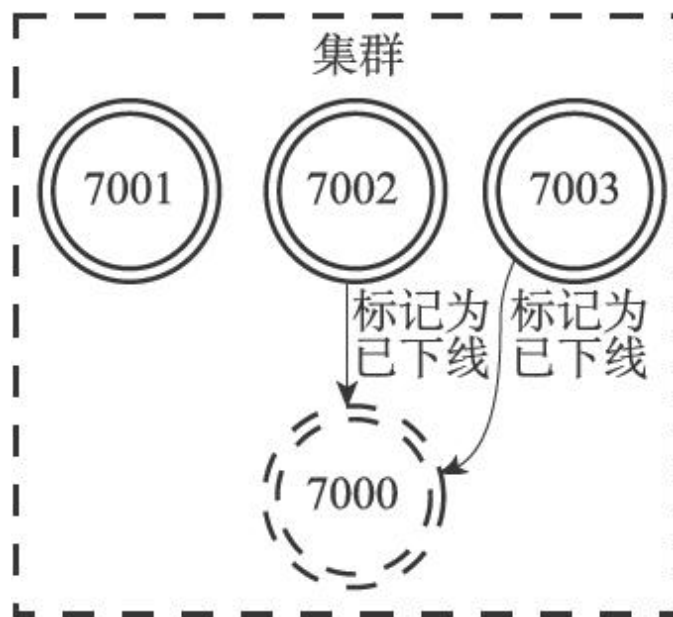


图17-44 节点7002和7003向节点7000发送下线消息

17.7.4 PUBLISH消息

□□□□□□□□□□□□□□□□

PUBLISH <channel> <message>

□□□□□□PUBLISH□□□□□□□□channel□□□□□□message□□□□
□□□□□□PUBLISH□□□□□□□□PUBLISH□□□□□□□□channel□□□□
message□□□

□□□□□□□□□□□□□□□□

PUBLISH <channel> <message>

□□□□□□□□□□□□channel□□□□message□□□

□□□□□□□□7000□7001□7002□7003□□□□□□□□□□□□7000
□□□□□□□□PUBLISH□□□□□□□7000□□7001□7002□7003□□□□□□
PUBLISH□□□□□17-45□□□

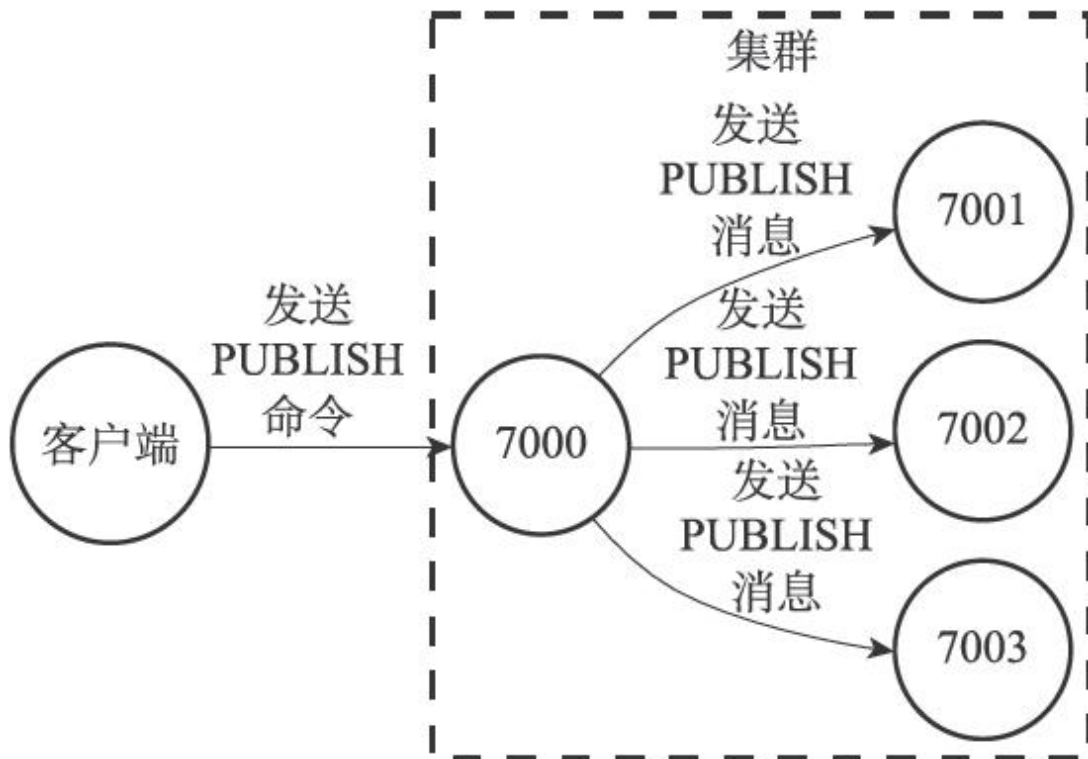


图17-45 向集群7000发送PUBLISH消息

PUBLISH消息结构定义在cluster.h/clusterMsgDataPublish中

```

typedef struct {
    uint32_t channel_len;
    uint32_t message_len;
    //
    // 8
    //
    unsigned char bulk_data[8];
} clusterMsgDataPublish;

```

clusterMsgDataPublish结构体bulk_data成员是一个长度为8的字节数组，用于存储PUBLISH消息的channel和message数据。

channel_len message_len channel message

· bulk_data[0] channel_len-1 channel

· bulk_data[channel_len] channel_len+message_len-1
message

PUBLISH

PUBLISH "news.it" "hello"

PUBLISH clusterMsgDataPublish 17-46
bulk_data channel "news.it" bulk_data
message "hello"

clusterMsgDataPublish											
channel_len											
7											
message_len											
5											
bulk_data											
→ 'n' 'e' 'w' 's' '.' 'i' 't' 'h' 'e' 'l' 'l' 'o'											

17-46 clusterMsgDataPublish

PUBLISH

□□□□□□□□□□□□□□PUBLISH□□□□□□□□□□□□□□□□
□PUBLISH□□□□□Redis□□PUBLISH□□□□□□□□□□□□□□□□
□Redis□□“□□□□□□□□□□□□□□□□”□□□□□□□□□□□□
PUBLISH□□□□□□

17.8 Redis

- Redis 是一个开源的分布式数据库

- Redis 支持 16384 个数据库，每个数据库都是一个独立的数据库，可以存储不同的数据

- Redis 支持主从复制，主节点负责写操作，从节点负责读操作，当主节点宕机时，从节点可以接管主节点的工作，保证数据的高可用性

- Redis 支持集群模式，redis-trib 是 Redis 的集群模式，可以实现分布式存储，支持高可用性和高扩展性

- Redis 支持 AOF 持久化，即 Append Only File，可以将 Redis 的写操作记录到文件中，当 Redis 重启时，可以从文件中恢复数据

- Redis 支持 Lua 脚本，可以在 Redis 中执行 Lua 脚本，实现复杂的业务逻辑

- Redis 支持多种数据类型，包括字符串、列表、集合、哈希等

- Redis 支持多种命令，包括 SET、GET、DEL、INCR、DECR、PUBLISH、SUBSCRIBE、UNSUBSCRIBE、PUBLISH、FAIL 等

□□□□ □□□□□□□□

□18□ □□□□

□19□ □□

□20□ Lua□□

□21□ □□

□22□ □□□□□□

□23□ □□□□□

□24□ □□□

18 消息队列

Redis 提供了 `PUBLISH`、`SUBSCRIBE`、`PSUBSCRIBE` 三个命令，用于实现消息队列。

`SUBSCRIBE` 命令用于订阅一个或多个频道。当有消息发布到订阅的频道时，Redis 会返回一个包含发布消息的数组。该数组的第一个元素是发布消息的频道名，第二个元素是发布消息的值。例如，如果订阅了 `news.it` 频道，并发布了消息 `hello`，那么返回的数组将是 `["news.it", "hello"]`。

假设我们有三个客户端 A、B、C，它们都订阅了 `news.it` 频道。

```
SUBSCRIBE "news.it"
```

当客户端 A 发布消息 `hello` 到 `news.it` 频道时，所有订阅该频道的客户端都会收到该消息。

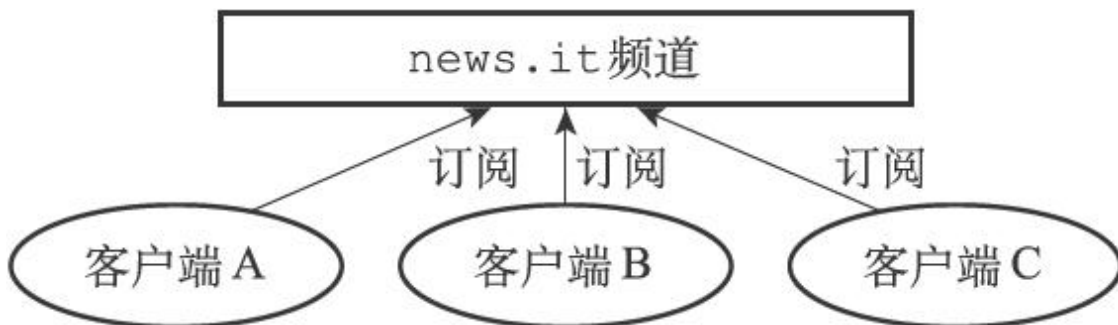


图 18-1 news.it 消息队列示意图

当客户端 A 发布消息 `hello` 到 `news.it` 频道时，所有订阅该频道的客户端都会收到该消息。

```
PUBLISH "news.it" "hello"
```

向"news.it"发布消息"hello"向"news.it"发布消息
18-2

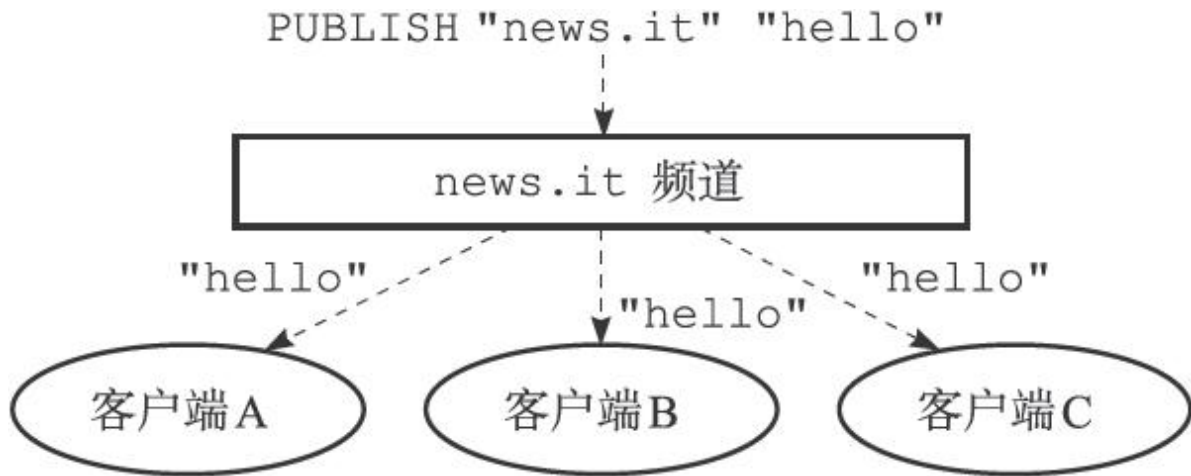


图18-2 发布消息

客户端订阅消息模式PSUBSCRIBE
客户端订阅消息模式PSUBSCRIBE
客户端订阅消息模式PSUBSCRIBE

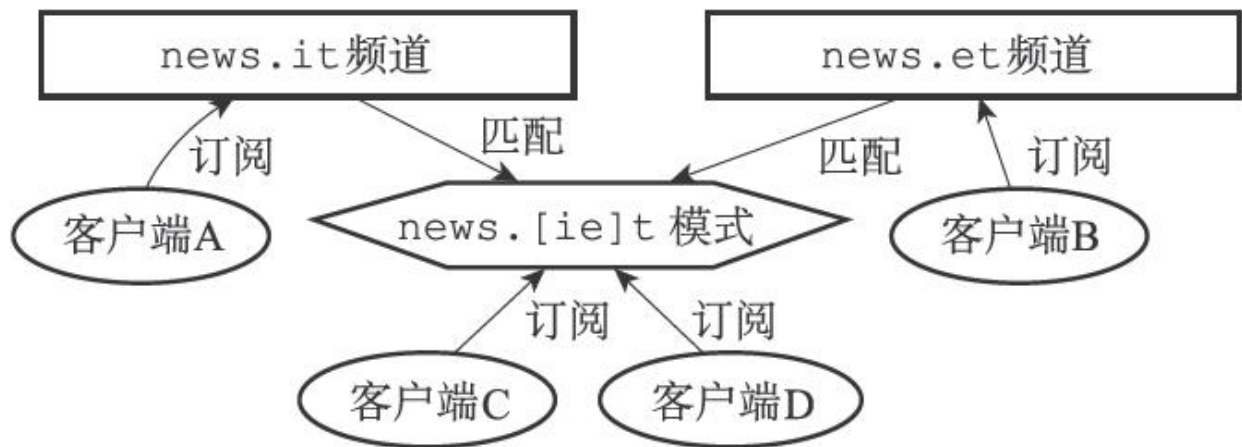


图18-3 消息匹配模式

□□□□□□□□18-3□□□

· 000A000000"news.it"

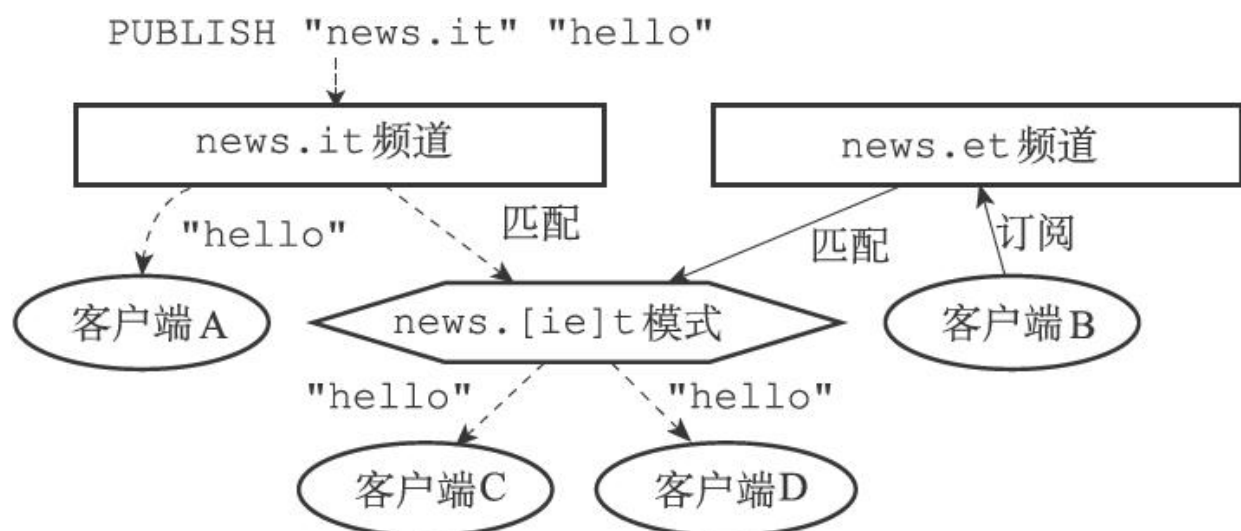
·B"news.et"

```
·CDD"news.it""news.et""news.
[ie]t"
```

□ □ □ □ □ □ □ □ □ □ □ □

```
PUBLISH "news.it" "hello"
```

"news.it"hello"news.it"A
 C D "news.it"news.
 [ie]t"18-4



□18-4 □□□□□□□□□□□□□□□□□□□□1□

□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □

"news.et" "world" "news.et" B
C D "news.et"
"news.[ie]t" 18-5

□18-5 □□□□□□□□□□□□□□□□□□□□□□□2□

PUBLISH

Redis 2.8 PUBSUB

18.1 发布订阅

Redis 提供了 SUBSCRIBE 命令，用于订阅指定的频道。订阅成功后，Redis 会将该客户端加入到指定的频道中。当有消息发布到该频道时，Redis 会将消息发送给所有订阅该频道的客户端。

Redis 使用 pubsub_channels 字典来管理发布订阅。该字典的键是频道名称，值是订阅该频道的客户端列表。当有消息发布到该频道时，Redis 会从该字典中取出所有订阅该频道的客户端，并将消息发送给它们。

```
struct redisServer {  
    // ...  
    //  
    dict *pubsub_channels;  
    // ...  
};
```

图 18-6 展示了 Redis 的 pubsub_channels 字典的结构。

·client-1 client-2 client-3 订阅了 "news.it" 频道

·client-4 订阅了 "news.sport" 频道

·client-5 client-6 订阅了 "news.business" 频道

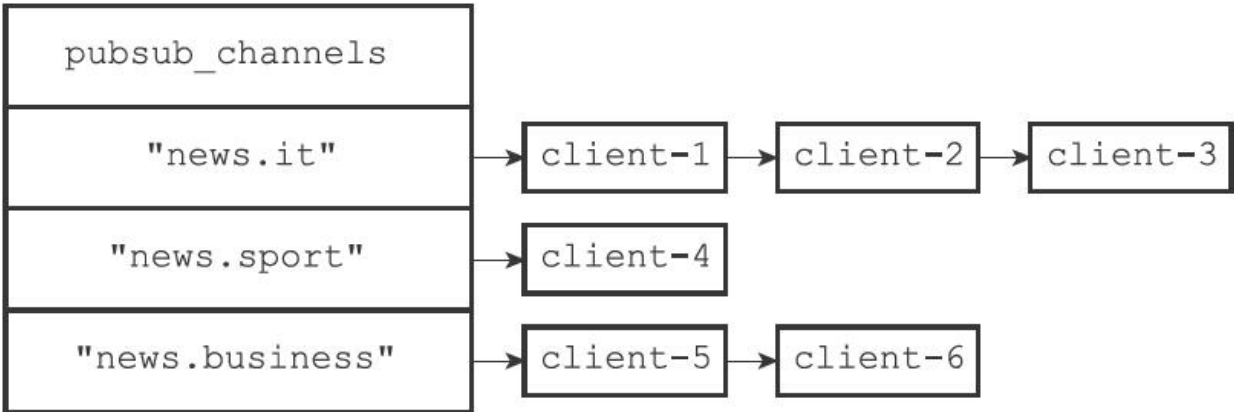


图18-6 pubsub_channels结构

18.1.1 订阅

当客户端发送SUBSCRIBE消息时，服务器会检查该消息中的channel名称是否在`pubsub_channels`列表中。

如果不在，服务器会返回错误。

· 如果channel名称在`pubsub_channels`列表中，服务器会检查该channel名称是否以`news.`开头。

· 如果channel名称以`news.`开头，服务器会检查该channel名称是否在`pubsub_channels`列表中。如果不在，服务器会返回错误。

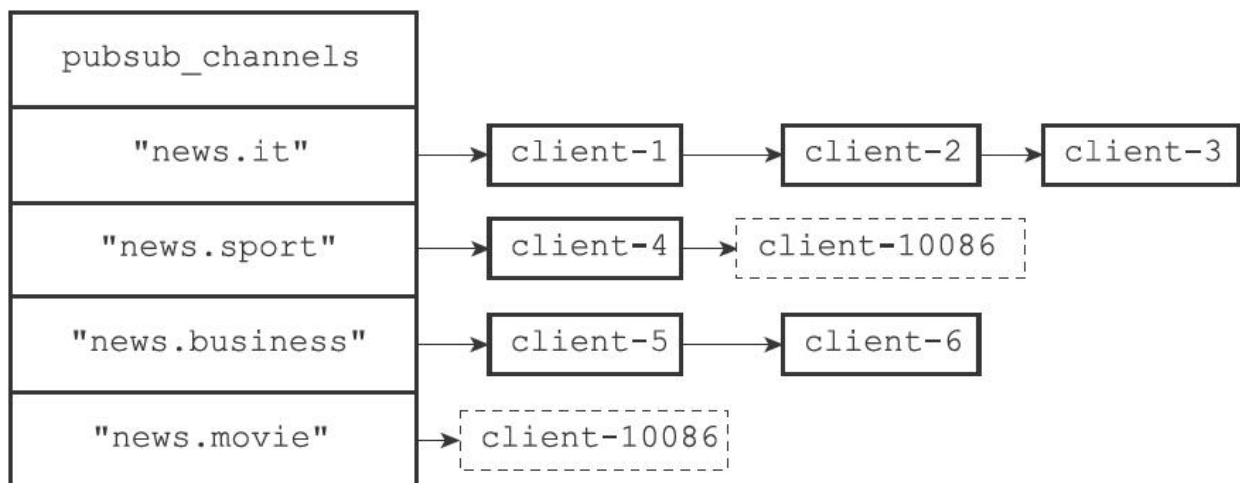
如果channel名称在`pubsub_channels`列表中，服务器会返回该channel名称对应的client ID列表。例如，对于`news.it`，返回的client ID列表是`client-10086`。

```
SUBSCRIBE "news.sport" "news.movie"
```

pubsub_channels18-7

·pubsub_channels"news.movie"
client-10086client-10086
"news.movie"

·"news.sport"client-10086
client-4



18-7 SUBSCRIBEpubsub_channels

SUBSCRIBE

```
def subscribe(*all_input_channels):  
    #  
    for channel in all_input_channels:
```



```

    #
    channel
    pubsub_channels
    #
    channel
    if channel not in server.pubsub_channels:
        server.pubsub_channels[channel] = []
    #
    server.pubsub_channels[channel].append(client)

```

18.1.2

UNSUBSCRIBE SUBSCRIBE

pubsub_channels

· pubsub_channels

·

pubsub_channels

pubsub_channels 18-8 client-10086

UNSUBSCRIBE "news.sport" "news.movie"

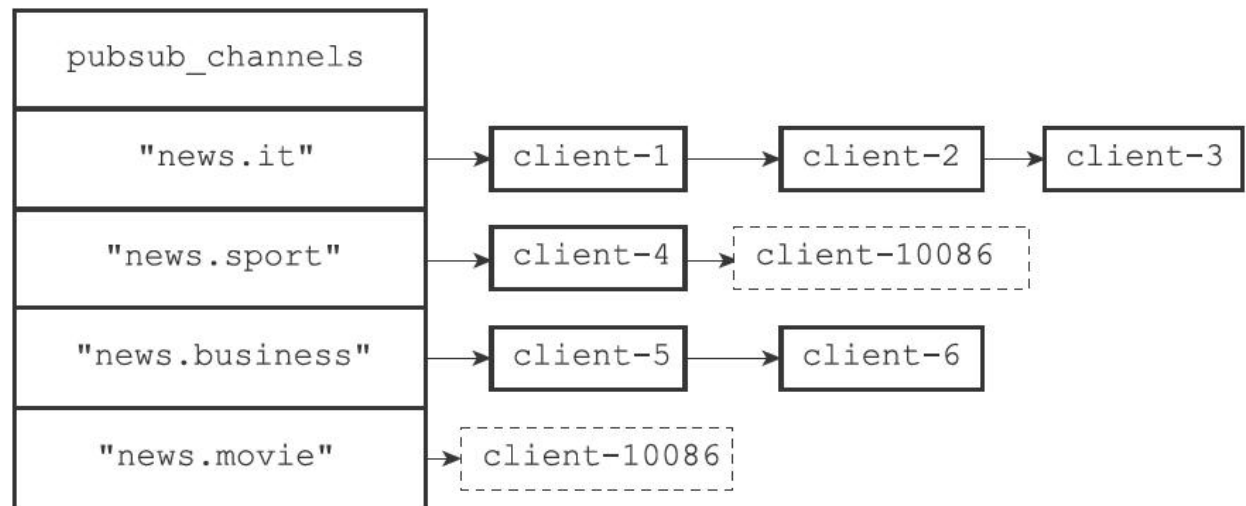
18-9

· pubsub_channels 10086

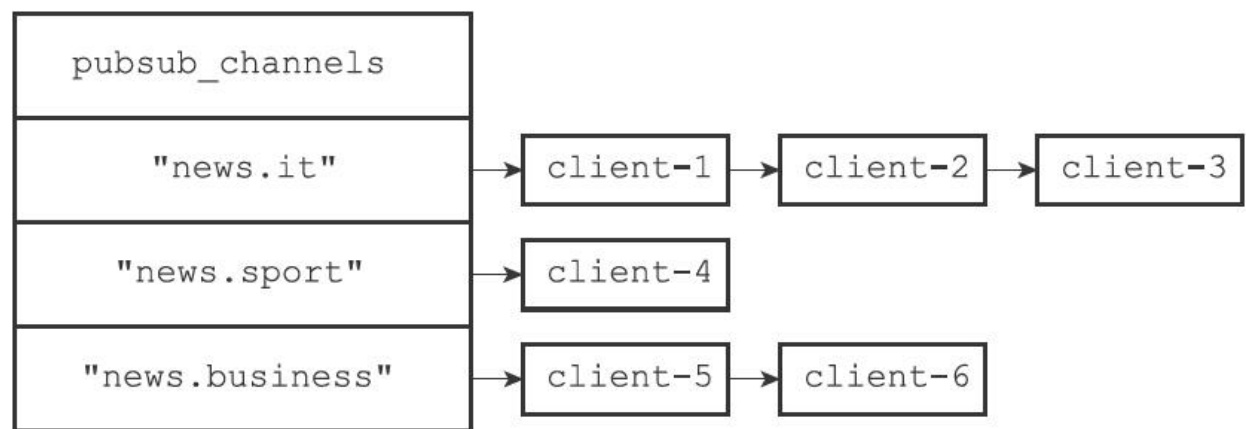
"news.sport" "news.movie"

· client-10086 "news.movie"

"news.movie"



18-8 UNSUBSCRIBE pubsub_channels



18-9 UNSUBSCRIBE pubsub_channels

UNSUBSCRIBE

```
def unsubscribe(*all_input_channels):
    #
    #####
    for channel in all_input_channels:
        #
        #####
        server.pubsub_channels[channel].remove(client)
        #
        #####
        #
        #####
        if len(server.pubsub_channels[channel]) == 0:
            server.pubsub_channels.remove(channel)
```

18.2 发布订阅

发布订阅功能由两个全局变量实现：`pubsub_channels` 和 `pubsub_patterns`

```
struct redisServer {  
    // ...  
    //  
    list *pubsub_patterns;  
    // ...  
};
```

`pubsub_patterns` 是一个链表，每个元素都是一个 `pubsubPattern` 结构体，它包含一个 `pattern` 字符串和一个 `client` 指针，指向订阅该模式的客户端。

```
typedef struct pubsubPattern {  
    //  
    redisClient *client;  
    //  
    robj *pattern;  
} pubsubPattern;
```

图 18-10 展示了 `pubsubPattern` 结构体与 `client-9` 的关系。

`"news.*"`

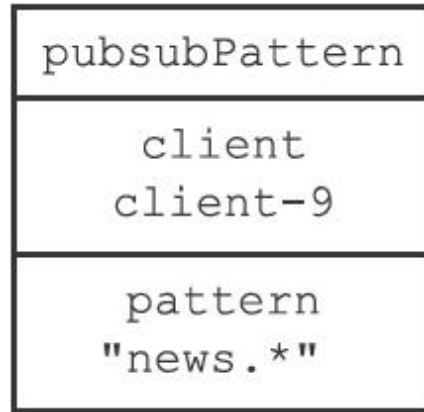


图18-10 pubsubPattern结构

图18-11展示了pubsub_patterns数据结构

· 客户端client-7订阅模式"music.*"

· 客户端client-8订阅模式"book.*"

· 客户端client-9订阅模式"news.*"

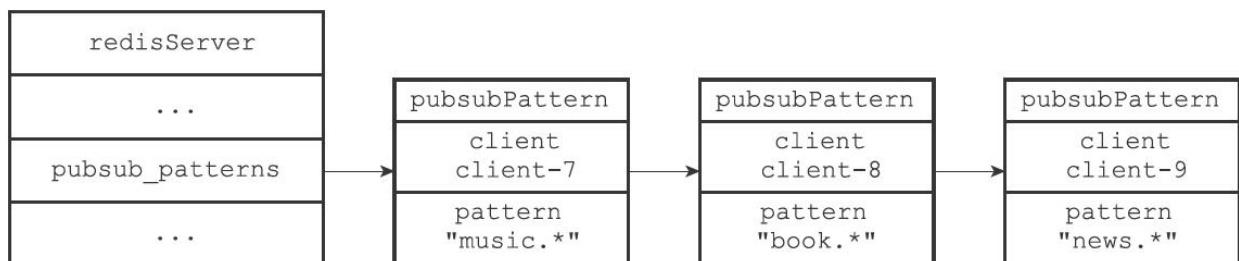


图18-11 pubsub_patterns结构

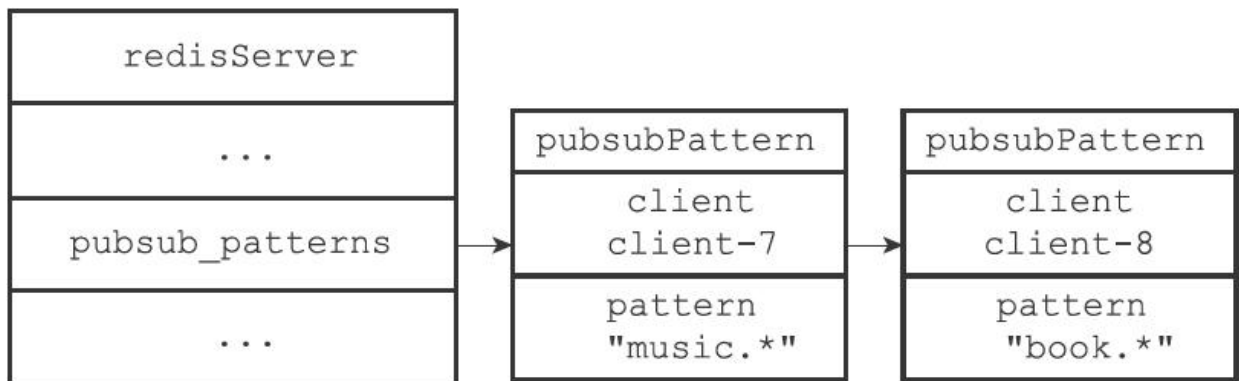
18.2.1 订阅

客户端通过PSUBSCRIBE命令订阅模式

命令格式

1 redisServer pubsubPattern pattern client
redisServer

2 redisServer pubsubPattern pubsub_patterns
redisServer pubsub_patterns 18-12

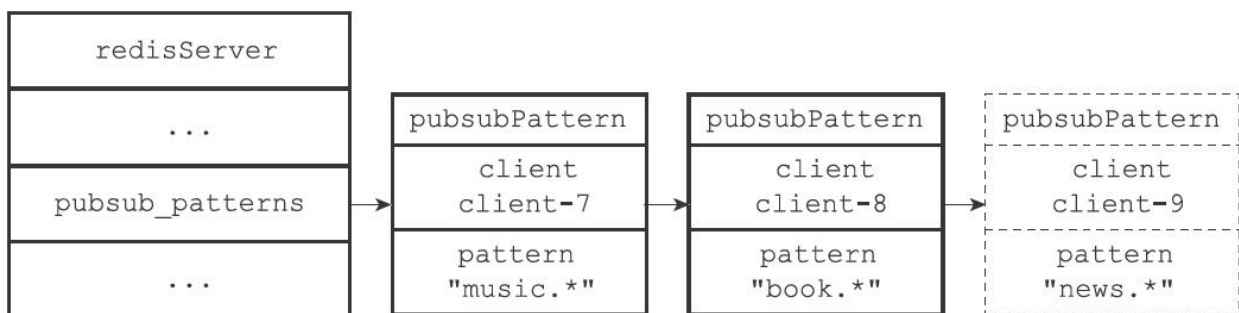


18-12 PSUBSCRIBE pubsub_patterns

client-9

PSUBSCRIBE "news.*"

redisServer pubsub_patterns 18-13
pubsubPattern



18-13 实现PSUBSCRIBE和pubsub_patterns

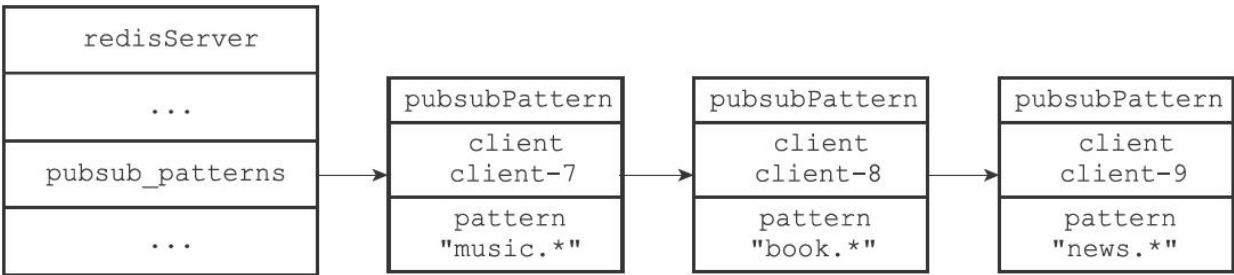
PSUBSCRIBE实现

```
def psubscribe(*all_input_patterns):
    #
    for pattern in all_input_patterns:
        #
        pubsubPattern
        #
        pubsubPattern = create_new_pubsubPattern()
        pubsubPattern.client = client
        pubsubPattern.pattern = pattern
        #
        pubsubPattern
        pubsub_patterns
        server.pubsub_patterns.append(pubsubPattern)
```

18.2.2 实现

实现PUNSUBSCRIBE和PSUBSCRIBE，实现pubsub_patterns，实现pattern，实现client，实现pubsubPattern。

实现pubsub_patterns，实现18-14。

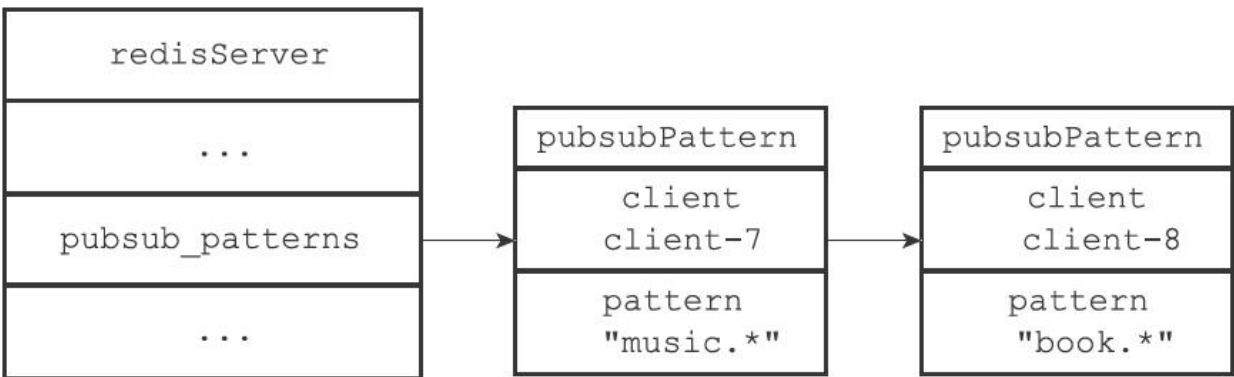


18-14 在PUNSUBSCRIBE之前从pubsub_patterns中

移除client-9

PUNSUBSCRIBE "news.*"

从client-9中移除pattern "news.*"的pubsubPattern
从pubsub_patterns中移除18-15



18-15 在PUNSUBSCRIBE之后从pubsub_patterns中

移除client-9

```

def punsubscribe(*all_input_patterns):
    #
    for pattern in all_input_patterns:
        #
  
```



```

pubsub_patterns
pubsubPattern
for pubsubPattern in server.pubsub_patterns:
    #
pubsubPattern
pubsubPattern
    #
pubsubPattern
pubsubPattern
    if client == pubsubPattern.client and \
        pattern == pubsubPattern.pattern:
        #
pubsubPattern
pubsubPattern
    server.pubsub_patterns.remove(pubsubPattern)

```

18.3 pubsub

Redis pubsub PUBLISH <channel> <message> 通过向 channel 发布消息

1 向 channel 发布消息

2 通过 pattern 匹配 channel 发布消息

Redis pubsub 消息发布与接收

18.3.1 pubsub 消息发布与接收

Redis pubsub 消息发布与接收 pubsub_channels 数据结构用于存储所有已发布的消息。channel 发布消息 PUBLISH 时，消息会被放入 pubsub_channels 数据结构。channel 发布消息时，消息会被放入 pubsub_channels 数据结构。pubsub_channels 数据结构在 18-16 中

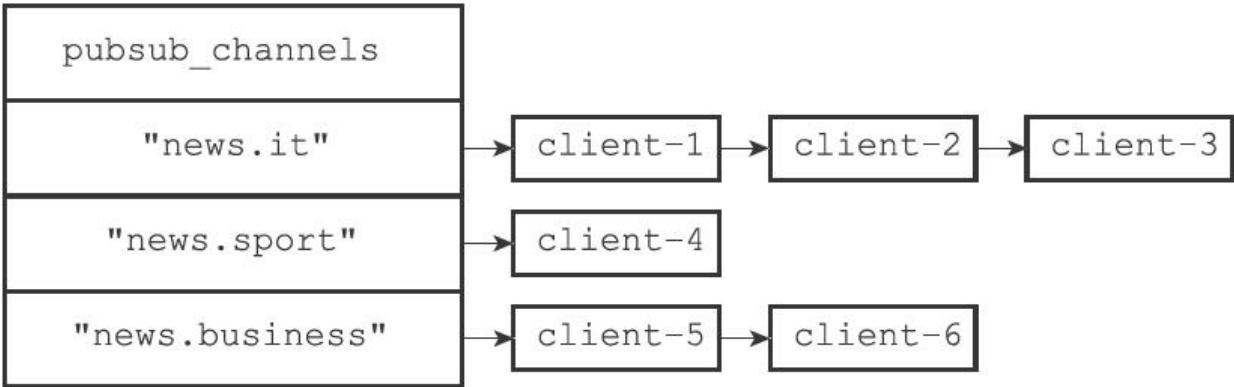


图18-16 pubsub_channels

消息发布

PUBLISH "news.it" "hello"

当PUBLISH消息到达pubsub_channels时，系统会遍历所有订阅该消息的客户端，并将消息发送给它们。例如，如果消息是"hello"，系统会遍历所有订阅"news.it"的客户端，并将消息发送给client-1、client-2和client-3。

PUBLISH消息的格式如下：

```
def channel_publish(channel, message):
    #
    channel = server.pubsub_channels[channel]
    #
    for client in channel:
        #
        if channel not in server.pubsub_channels:
            return
        #
```

```

    for subscriber in server.psub_channels[channel]:
        send_message(subscriber, message)

```

18.3.2 发布消息

Redis 的发布/订阅功能是通过 `pubsub_patterns` 字典来管理的。当客户端向 Redis 发送 `PUBLISH` 命令时，Redis 会根据消息的频道名称，在 `pubsub_patterns` 字典中找到对应的频道，并将消息发送给该频道的所有订阅者。

图 18-17 展示了 `pubsub_patterns` 字典的结构。

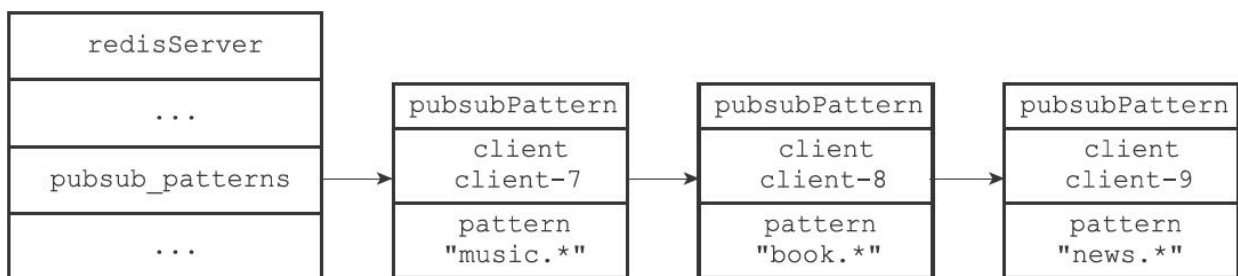


图 18-17 pubsub_patterns 字典结构

当客户端向 Redis 发送 `PUBLISH` 命令时，Redis 会根据消息的频道名称，在 `pubsub_patterns` 字典中找到对应的频道，并将消息发送给该频道的所有订阅者。

```

PUBLISH "news.it" "hello"

```

```
    PUBLISH("hello","news.it")

pubsub_patterns["news.it"]

"news.it"client-9"news.*"hello"

client-9
```

PUBLISH

```
def pattern_publish(channel, message):
    #
    for pubsubPattern in server.pubsub_patterns:
        #
        if match(channel, pubsubPattern.pattern):
            #
            send_message(pubsubPattern.client, message)
```

PUBLISH

```
def publish(channel, message):
    #
    channel
    channel_publish(channel, message)
    #
    channel
    pattern_publish(channel, message)
```

18.4 发布订阅

PUBSUB 是 Redis 2.8 版本引入的一个特性，它允许你发布消息到指定的频道，并订阅感兴趣的频道。当有消息发布到你所订阅的频道时，你会收到消息。

发布订阅功能由两个命令组成：PUBSUB 和 CHANNELS。

18.4.1 PUBSUB CHANNELS

PUBSUB CHANNELS[pattern] 命令用于查看当前订阅的频道列表。如果提供了 pattern，则只返回匹配该模式的频道。

· 不带 pattern 参数，返回所有当前订阅的频道。

· 带 pattern 参数，返回匹配该模式的频道。

例如：

```
pubsub_channels  # 查看所有订阅的频道
pubsub_channels pattern  # 查看匹配 pattern 的订阅频道
```

```
def pubsub_channels(pattern=None):
    #
    # 初始化频道列表
    channel_list = []
    #
    # 遍历所有频道
    #
    # 返回 pubsub_channels
```

```

def _find_channel(server, pattern):
    for channel in server.pubsub_channels:
        #
        #1
        if pattern:
            #2
            if match(channel, pattern):
                channel_list.append(channel)
    #
    return channel_list

```

图 18-18 pubsub_channels 数据结构

PUBSUB CHANNELS 命令返回的结果

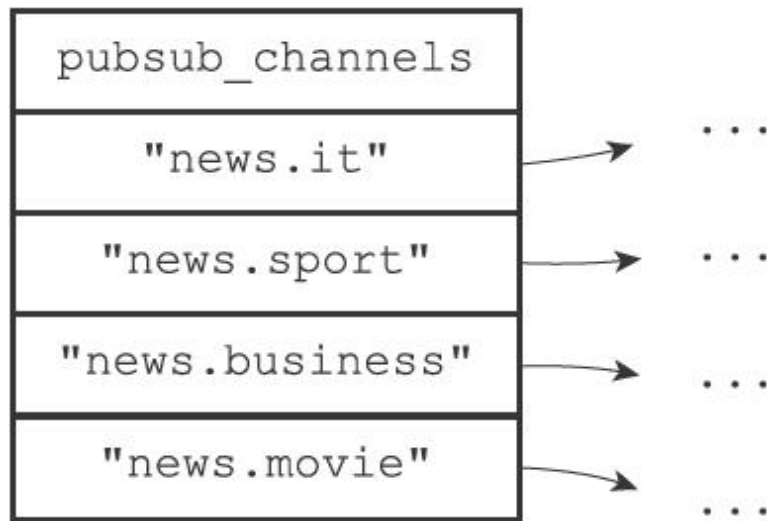


图 18-18 pubsub_channels 数据结构

```

redis> PUBSUB CHANNELS
1) "news.it"
2) "news.sport"

```

- 3) "news.business"
- 4) "news.movie"

```
PUBSUB CHANNELS "news.[is]*"
"news.it" "news.sport" "news.[is]*"
```

```
redis> PUBSUB CHANNELS "news.[is]*"
1) "news.it"
2) "news.sport"
```

18.4.2 PUBSUB NUMSUB

```
PUBSUB NUMSUB[channel-1 channel-2...channel-n]
[...]
```

```
pubsub_channels[...]
```

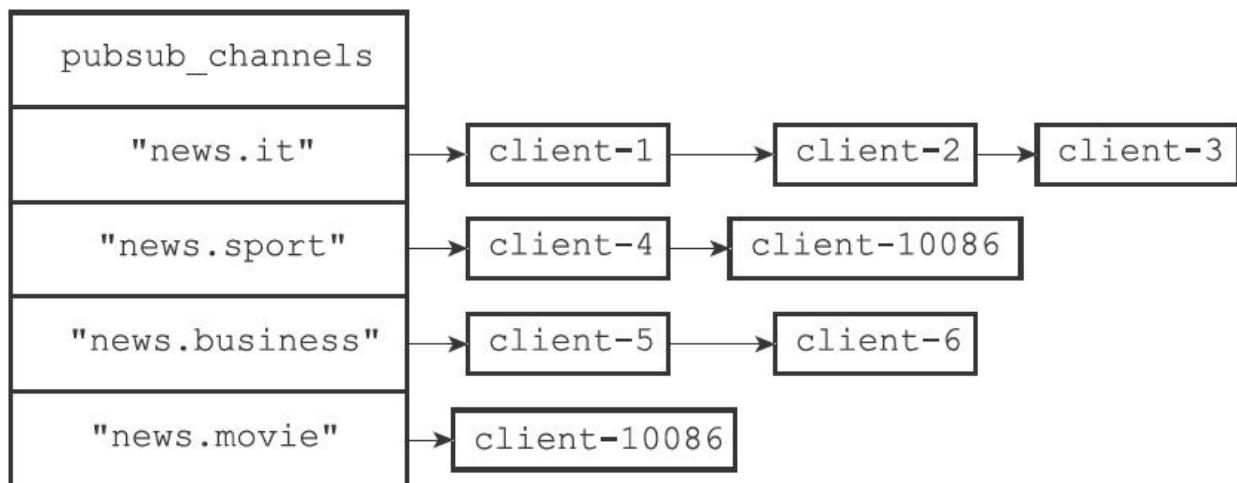
```
def pubsub_numsub(*all_input_channels):
    #
    for channel in all_input_channels:
        #
        pubsub_channels
        channel
        #
        channel
        if channel not in server.pubsub_channels:
            #
            reply_channel_name(channel)
            #
```



```

        0
        reply_subscribe_count(0)
        #
        pubsub_channels
        channel
        #
        channel
        else:
            #
            reply_channel_name(channel)
            #
            reply_subscribe_count(len(server.pubsub_channels[channel]))

```



18-19 pubsub_channels

18-19 pubsub_channels

PUBSUB NUMSUB

```

redis> PUBSUB NUMSUB news.it news.sport news.business news.movie
1) "news.it"
2) "3"
3) "news.sport"
4) "2"
5) "news.business"

```

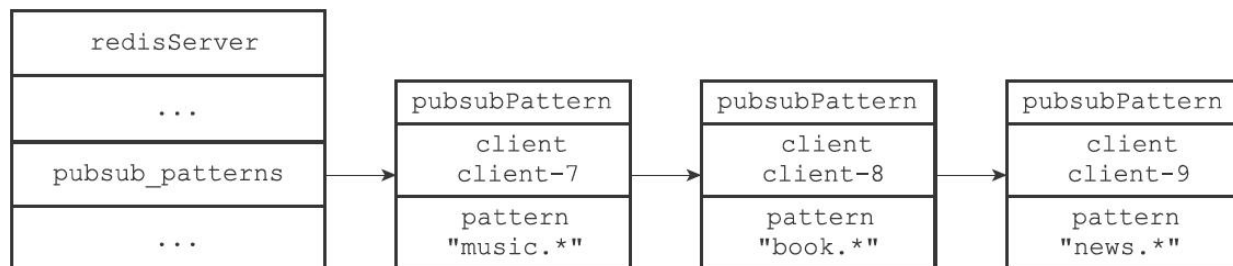
```
6) "2"
7) "news.movie"
8) "1"
```

18.4.3 PUBSUB NUMPAT

PUBSUB NUMPAT

pubsub_patterns

```
def pubsub_numpat():
    # pubsub_patterns
    reply_pattern_count(len(server.pubsub_patterns))
```



18-20 pubsub_patterns

18-20 pubsub_patterns PUBSUB NUMPAT

```
redis> PUBSUB NUMPAT
(integer) 3
```

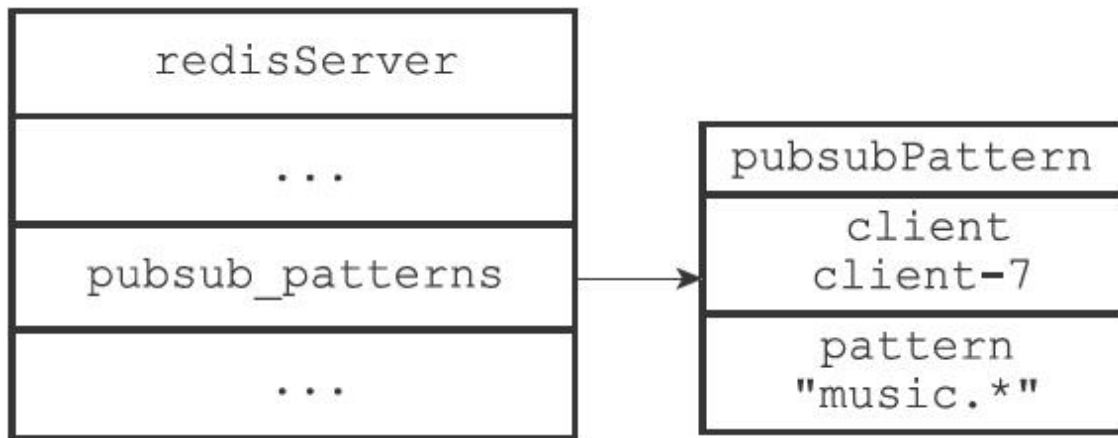


图18-21 pubsub_patterns

图18-21中的pubsub_patterns指向的PUBSUB NUMPAT

返回1

```
redis> PUBSUB NUMPAT
(integer) 1
```

18.5 消息队列

· 向消息队列pubsub_channels发送消息时，消息格式为SUBSCRIBE消息，消息格式如下：

UNSUBSCRIBE消息格式如下：

· 向消息队列pubsub_patterns发送消息时，消息格式为PSUBSCRIBE消息，消息格式如下：

PUNSUBSCRIBE消息格式如下：

· PUBLISH消息格式如下：

pubsub_patterns消息格式如下：

· PUBSUB消息格式如下：

pubsub_patterns消息格式如下：

18.6 参考

- [Publish Subscribe Pattern](http://en.wikipedia.org/wiki/Publish_subscribe_pattern)

http://en.wikipedia.org/wiki/Publish_subscribe_pattern

5.7

- [Pattern-Oriented Software Architecture Volume 4](#)

[Pattern Language for Distributed Computing](#)

[Distribution Infrastructure](#)

- [Glob](http://en.wikipedia.org/wiki/Glob_programming)

http://en.wikipedia.org/wiki/Glob_programming

[glob](#) 7 [Wildcard Matching](#)

19 節

Redis の MULTI EXEC WATCH による transaction

Redis の transaction 機能は、複数の Redis 命令を原子的に実行するための機能です。これにより、複数の命令が同時に実行されることを保証し、データの整合性を保つことができます。

Redis の transaction 機能は、MULTI 命令で開始されます。

EXEC 命令で commit されます。

```
redis> MULTI
OK
redis> SET "name" "Practical Common Lisp"
QUEUED
redis> GET "name"
QUEUED
redis> SET "author" "Peter Seibel"
QUEUED
redis> GET "author"
QUEUED
redis> EXEC
1) OK
2) "Practical Common Lisp"
3) OK
4) "Peter Seibel"
```

Redis の MULTI EXEC による transaction

Redis の transaction 機能は、複数の Redis 命令を原子的に実行するための機能です。

WATCHWATCH

ACIDRedis

19.1 多路复用

Redis 1.2 版本开始支持多路复用

1. 多路复用

2. 多路复用

3. 多路复用

Redis 1.2 版本开始支持多路复用

19.1.1 多路复用

MULTI 命令

```
redis> MULTI
OK
```

MULTI 命令

flags 标志位 REDIS_MULTIMASTER 标志位 MULTI 标志位

```
def MULTI():
    #
    # 标志位
    client.flags |= REDIS_MULTIMASTER
    #
    return OK
```


replyOK()

19.1.2 数据库

数据库

```
redis> SET "name" "Practical Common Lisp"
OK
redis> GET "name"
"Practical Common Lisp"
redis> SET "author" "Peter Seibel"
OK
redis> GET "author"
"Peter Seibel"
```

数据库

数据库

·数据库EXEC·DISCARD·WATCH·MULTI数据库

数据库

·数据库EXEC·DISCARD·WATCH·MULTI数据库

数据库

QUEUED

数据库19-1数据库

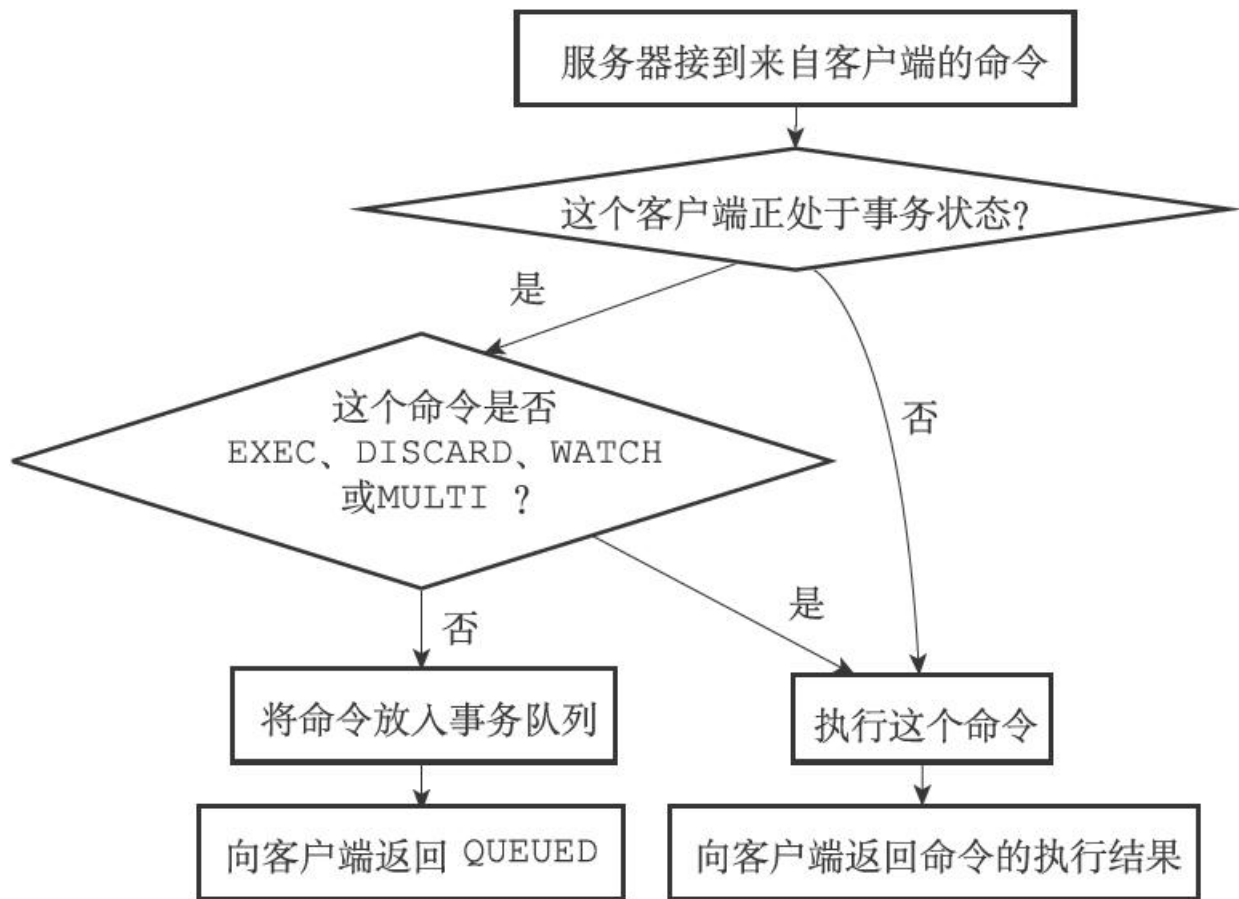


图19-1 Redis事务逻辑流程图

19.1.3 Redis事务

Redis事务是通过客户端发送的MULTI命令来开启的，事务状态通过mstate来维护。

```

typedef struct redisClient {
    // ...
    //
    multiState mstate; /* MULTI/EXEC state */
    // ...
} redisClient;
  
```

redis> MULTI

```
typedef struct multiState {  
    //  
    redisFIFO  
    //  
    multiCmd *commands;  
    //  
    int count;  
} multiState;
```

redis> MULTI
OK
redis> SET "name" "Practical Common Lisp"

```
typedef struct multiCmd {  
    //  
    robj **argv;  
    //  
    int argc;  
    //  
    struct redisCommand *cmd;  
} multiCmd;
```

redis> MULTI
OK
redis> SET "name" "Practical Common Lisp"

redis> MULTI

```
redis> MULTI  
OK  
redis> SET "name" "Practical Common Lisp"
```

```

QUEUED
redis> GET "name"
QUEUED
redis> SET "author" "Peter Seibel"
QUEUED
redis> GET "author"
QUEUED

```

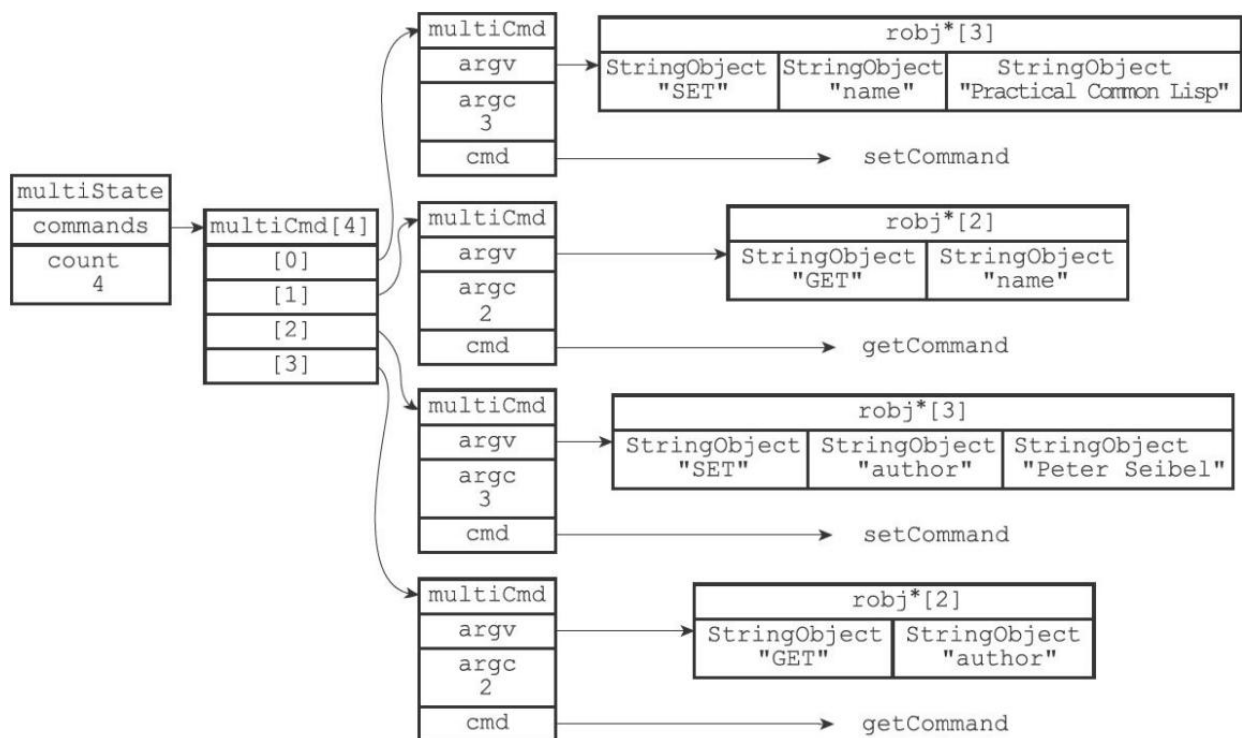
19-2

• SET 0

• GET 1

• SET 2

• GET 3



19-2 練習

19.1.4 練習

この練習では、`EXEC` を使って、`EXEC` の実行結果を
表示するプログラムを作成します。

この練習は 19-2 の練習の一部です。

```
SET "name" "Practical Common Lisp"
```

```
;;
```

```
GET "name"
```

```
;;
```

```
SET "author" "Peter Seibel"
```

```
;;
```

```
GET "author"
```

```
;;
```

```
redis> EXEC
1) OK
2) "Practical Common Lisp"
3) OK
4) "Peter Seibel"
```

EXEC

```
def EXEC():
    #
    # Create a queue to hold replies
    reply_queue = []
    #
    # Iterate over the commands in the transaction
    for argv, argc, cmd in client.mstate.commands:
        #
        # Execute the command
        reply = execute_command(cmd, argv, argc)
        #
        # Append the reply to the queue
        reply_queue.append(reply)
        #
    # If Redis MULTI was used, clear the MULTI flag
    client.flags &= ~REDIS_MULTI
    #
    # If Redis MULTI was used, clear the MULTI flag
    #1
    #2
    # Clear the transaction count
    client.mstate.count = 0
    release_transaction_queue(client.mstate.commands)
    #
    # Send the replies back to the client
    send_reply_to_client(client, reply_queue)
```

19.2 WATCH

WATCH 实现 optimistic locking EXEC 实现

EXEC 实现

```
redis> WATCH "name"
OK
redis> MULTI
OK
redis> SET "name" "peter"
QUEUED
redis> EXEC
(nil)
```

图 19-1 乐观锁实现

图 19-1 乐观锁实现

时间	客户端 A	客户端 B
T1	WATCH "name"	
T2	MULTI	
T3	SET "name" "peter"	
T4		SET "name" "john"
T5	EXEC	

在T4之前B的"name"属性值A在T5之前EXEC之前
WATCH之前"name"属性值A在A之前

在WATCH之前
在

19.2.1 WATCH

Redis的watched_keys WATCH
在

```
typedef struct redisDb {  
    // ...  
    //  
    WATCH  
    dict *watched_keys;  
    // ...  
} redisDb;
```

watched_keys
在

19-3 watched_keys watched_keys
在

·c1c2"name"

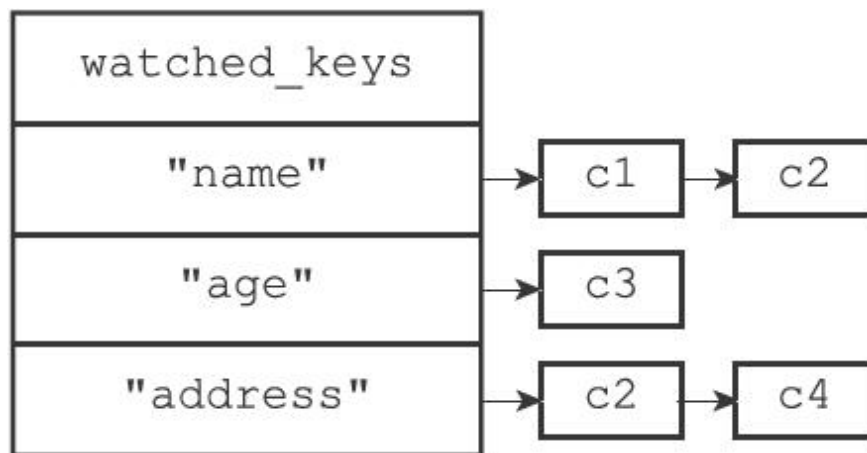
·c3"age"

·c2c4"address"

WATCHwatched_keys
c10086WATCH

```
redis> WATCH "name" "age"
OK
```

19-3watched_keys19-4
c10086WATCH



19-3 watched_keys

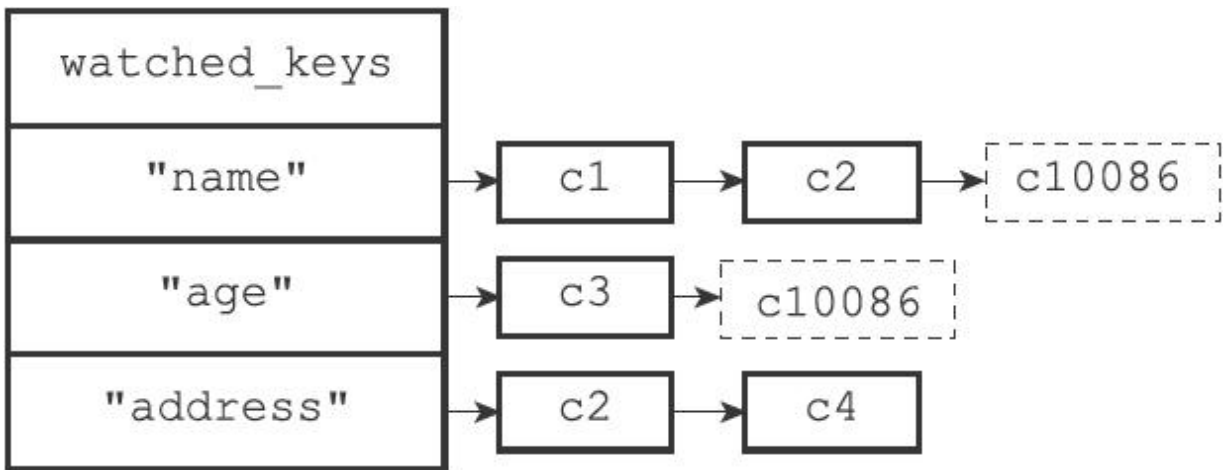


图19-4 WATCH的watched_keys

19.2.2 实现原理

Redis支持WATCH的原子命令有SET、LPUSH、SADD、ZREM、DEL、FLUSHDB。实现原理可以参考[multi.c/touchWatchKey](#)。watched_keys是哈希表，键是key，值是watched_keys的指针。touchWatchKey函数将key的value设置为REDIS_DIRTY_CAS，表示该key被修改。

touchWatchKey函数实现如下：

```
def touchWatchKey(db, key):
    #
    key = key
    watched_keys = db.watched_keys
    #
    # key
    if key in db.watched_keys:
        #
        key = key
        #
```

```

for client in db.watched_keys[key]:
    #
    client.flags |= REDIS_DIRTY_CAS

```

图 19-5 watched_keys

```

·{"name":c1,c2,c10086}
REDIS_DIRTY_CAS

·{"age":c3,c10086}REDIS_DIRTY_CAS

·{"address":c2,c4}REDIS_DIRTY_CAS

```

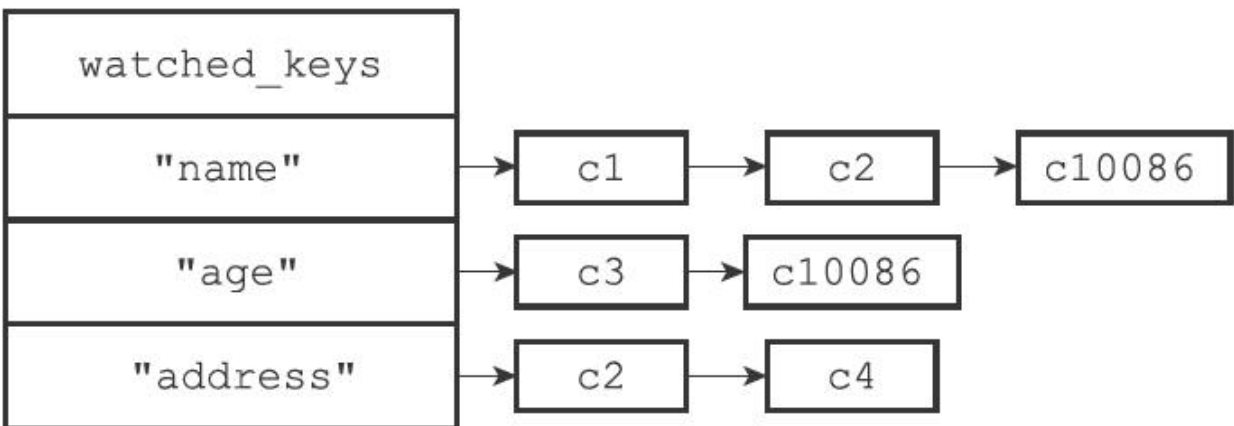


图 19-5 watched_keys

19.2.3 数据库

客户端向服务器发送EXEC命令
Redis_Dirty_Cas标识是否已经打开

·Redis_Dirty_Cas标识是否已经打开
如果已经打开，则客户端提交的事务
如果未打开，则客户端提交的事务

·Redis_Dirty_Cas标识是否已经打开
如果已经打开，则客户端提交的事务
如果未打开，则客户端提交的事务

图19-6 客户端提交事务流程图

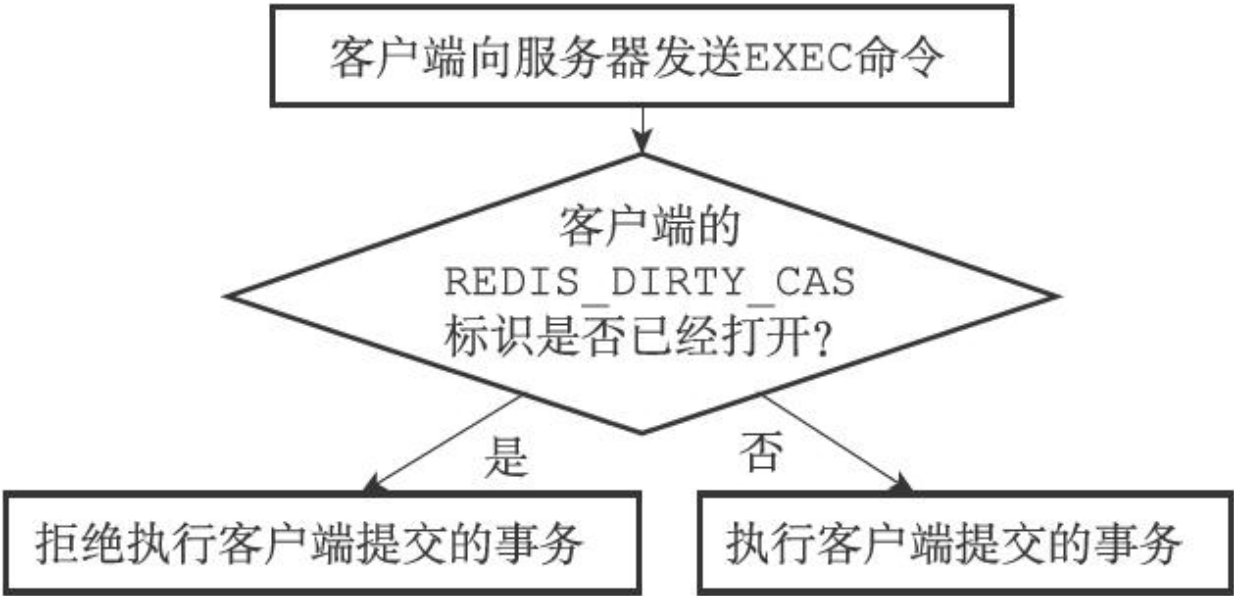


图19-6 客户端提交事务流程图

图19-5展示了watched_keys数组中"name"键
SET"name"john命令，c1和c2的c10086

REDIS_DIRTY_CAS 00000000000000000000000000000000 EXEC 0000000000000000
00000000000000000000000000000000

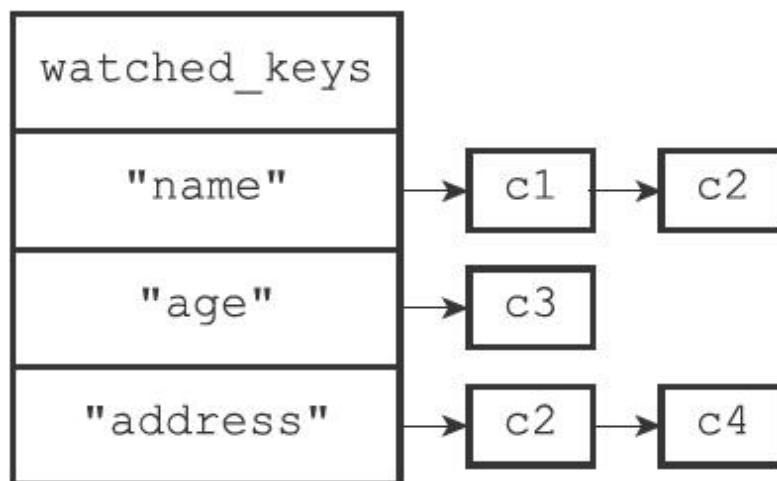
19.2.4 00000WATCH000000

0000000WATCH000000000000000000000000WATCH000000000000
000

00000000c1008600000watched_keys0000000019-70000
00c100860000WATCH00000

```
c10086> WATCH "name"  
OK
```

watched_keys000000019-8000000



019-7 00WATCH00000watched_keys00

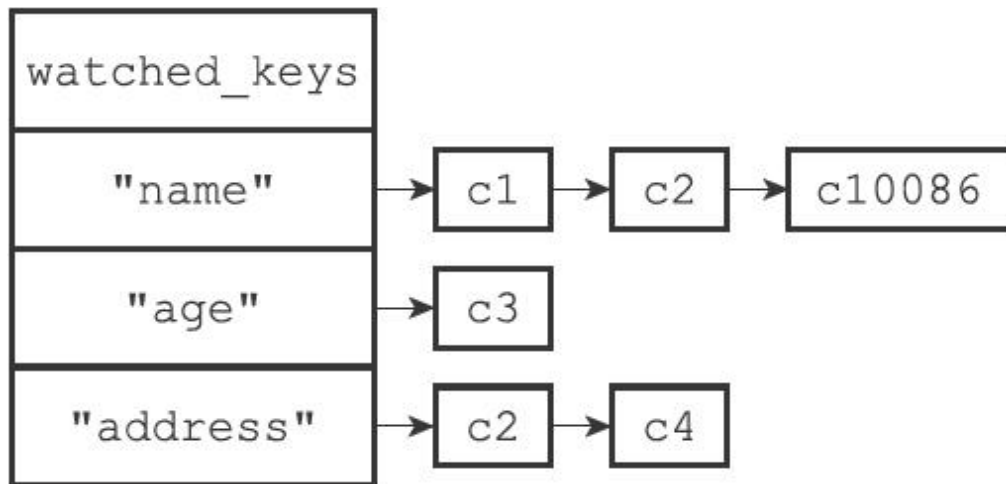


图19-8 WATCH操作的watched_keys

客户端c10086向服务器发送MULTI命令开始事务，然后发送SET命令

```

c10086> MULTI
OK
c10086> SET "name" "peter"
QUEUED
  
```

客户端c999向服务器发送SET命令设置"name"字段的值

"john"

```

c999> SET "name" "john"
OK
  
```

c999向服务器发送SET命令设置"name"字段的值

REDIS_DIRTY_CAS标志，表示c10086

c10086 EXEC c10086

REDIS_DIRTY_CAS

c10086> EXEC
(nil)

19.3 聊聊ACID

数据库系统通常都支持ACID

Redis支持AtomicityConsistencyIsolationRedis不支持Durability

数据库系统通常都支持ACID

19.3.1 聊聊

数据库系统通常都支持ACID
数据库系统通常都支持ACID

Redis支持AtomicityConsistencyIsolationRedis不支持Durability

数据库系统通常都支持ACID

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> GET msg
QUEUED
redis> EXEC
```


- 1) OK
- 2) "hello"

redis> SET msg "hello"

QUEUED

redis> GET

(error) ERR wrong number of arguments for 'get' command

redis> GET msg

QUEUED

redis> EXEC

(error) EXECABORT Transaction discarded because of previous errors.

Redis> MULTI

OK

redis> SET msg "hello"

QUEUED

redis> GET

(error) ERR wrong number of arguments for 'get' command

redis> GET msg

QUEUED

redis> EXEC

(error) EXECABORT Transaction discarded because of previous errors.

Redis> RPUSH msg "good bye" "bye bye" #

QUEUED

redis> EXEC

OK

redis> SET msg "hello" # msg

QUEUED

OK

redis> MULTI

OK

redis> SADD fruit "apple" "banana" "cherry"

QUEUED

redis> RPUSH msg "good bye" "bye bye" #

QUEUED

redis> EXEC

OK

3) (integer) 3

Redis

Redis

Redis

[illegible]

```
Redis[REDACTED]Redis
[REDACTED]Redis[REDACTED]
```

Redis

redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'

redis> GET msg
QUEUED
redis> EXEC
(error) EXECABORT Transaction discarded because of previous errors.

redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'

redis> GET msg
QUEUED
redis> EXEC
(error) EXECABORT Transaction discarded because of previous errors.

Redis 2.6.5

redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'

redis> GET msg
QUEUED
redis> EXEC
(error) EXECABORT Transaction discarded because of previous errors.

redis> SET msg "hello"
OK
redis> GET msg
hello

redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'

redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> YAHOOOO
(error) ERR unknown command 'YAHOOOO'

```
redis> EXEC
1) OK
2) "hello"
```

このチュートリアルではRedisのインストールと2.6.5の
インストールRedisのインストールと

2. インストール

このチュートリアルではRedisのインストールと

インストールRedisのインストールと

・インストールRedisのインストールと

インストールRedisのインストールと

・インストールRedisのインストールと

インストールRedisのインストールと

インストールRedisのインストールと

インストールRedisのインストールとSETコマンド"msg"のインストールと

インストールRedisのインストールとRPUSHコマンドのインストールと

インストールRedisのインストールと

```
redis> SET msg "hello"
OK
redis> MULTI
OK
redis> SADD fruit "apple" "banana" "cherry"
QUEUED
redis> RPush msg "good bye" "bye bye"
QUEUED
redis> SADD alphabet "a" "b" "c"
QUEUED
redis> EXEC
1) (integer) 3
2) (error) WRONGTYPE Operation against a key holding the wrong kind of
value
3) (integer) 3
```

Redis 3.0.0 版本开始支持事务，事务的语法与 MySQL 类似，使用 MULTI 命令开始事务，EXEC 命令结束事务。

3. Redis 事务

Redis 事务的语法与 MySQL 类似，使用 MULTI 命令开始事务，EXEC 命令结束事务。

Redis 事务的语法与 MySQL 类似，使用 MULTI 命令开始事务，EXEC 命令结束事务。

Redis 事务的语法与 MySQL 类似，使用 MULTI 命令开始事务，EXEC 命令结束事务。

Redis 19.3.3

Redis 19.3.3

Redis

19.3.3 Redis

Redis 19.3.3

Redis 19.3.3

Redis 19.3.3

Redis 19.3.3

19.3.4 Redis

Redis 19.3.4

Redis 19.3.4

Redis 19.3.4

Redis 19.3.4

Redis 19.3.4

Redis

· `no-appendfsync-on-rewrite` 选项禁止在 `RDB` 快照生成过程中调用 `BGSAVE` 或 `BGSAVE` 命令，因此快照生成期间不会调用 `RDB` 快照生成命令。

· `no-appendfsync-on-rewrite` 选项禁止在 `appendfsync` 命令中调用 `always` 选项，因此快照生成期间不会调用 `sync` 命令，因此快照生成期间不会调用 `sync` 命令。

· `no-appendfsync-on-rewrite` 选项禁止在 `appendfsync` 命令中调用 `everysec` 选项，因此快照生成期间不会调用 `sync` 命令，因此快照生成期间不会调用 `sync` 命令。

· `no-appendfsync-on-rewrite` 选项禁止在 `appendfsync` 命令中调用 `no` 选项，因此快照生成期间不会调用 `sync` 命令，因此快照生成期间不会调用 `sync` 命令。

no-appendfsync-on-rewrite 选项

`no-appendfsync-on-rewrite` 选项禁止在 `appendfsync` 命令中调用 `always`、`everysec`、`AOF` 或 `no-appendfsync-on-rewrite` 选项，因此快照生成期间不会调用 `BGSAVE` 或 `BGREWRITEAOF` 命令，因此快照生成期间不会调用 `AOF` 命令，因此快照生成期间不会调用 `I/O` 命令，因此快照生成期间不会调用 `“always”` 或 `AOF` 命令，因此快照生成期间不会调用 `AOF` 命令。

no-appendfsync-on-rewrite always AOF no-appendfsync-on-rewrite

Redis SAVE

```
redis> MULTI
OK
redis> SET msg "hello"
QUEUED
redis> SAVE
QUEUED
redis> EXEC
1) OK
2) OK
```


19.4 □□□□

- 数据持久化策略
- 数据同步策略
- 数据备份策略
- WATCH 命令
- REDIS_DIRTY_CAS 命令
- Redis ACID 模型
- Redis AOF 模型
- Redis appendfsync 策略

19.5 □□□□

- 弱酸性ACID 弱酸性ACID

<http://en.wikipedia.org/wiki/ACID>

- □□□□□□□□□□6□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- ```
·RedisRedisRedis
```

□□□□http://redis.io/topics/transactions□

## 20 Lua

Redis 2.6 开始支持 Lua 脚本。Lua 脚本在 Redis 中通过 EVAL 命令执行。

EVAL 命令的语法如下：

---

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

---

EVALSHA 命令用于执行已经通过 SHA1 哈希值存储在 Redis 中的 Lua 脚本。EVALSHA 命令的语法如下：

---

```
redis> EVAL "return 1+1" 0
(integer) 2
redis> EVALSHA "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9" 0 //
SHA1 哈希值
(integer) 2
```

---

SCRIPT LOAD 命令用于将 Lua 脚本加载到 Redis 中。

---

```
redis> SCRIPT LOAD "return 2*2"
"4475bfb5919b5ad16424cb50f74d4724ae833e72"
redis> EVALSHA "4475bfb5919b5ad16424cb50f74d4724ae833e72" 0
(integer) 4
```

---

Redis 支持 Lua 脚本的多个特性。

Redis Lua Redis Lua  
Lua

Lua Lua Redis  
Lua Redis  
Lua SCRIPT EXISTS

EVAL EVALSHA Lua Redis  
——SCRIPT FLUSH SCRIPT  
EXISTS SCRIPT LOAD SCRIPT KILL

Redis Lua

## 20.1 在Redis中运行Lua脚本

Redis提供了Lua脚本运行环境，可以在Redis中运行Lua脚本。Redis的Lua脚本运行环境是一个嵌入式的Lua解释器，可以在Redis中运行Lua脚本。

Redis提供了Lua脚本运行环境，可以在Redis中运行Lua脚本。

1. 在Redis中运行Lua脚本

2. 在Redis中运行Lua脚本

3. 在Redis中运行Lua脚本

4. 在Redis中运行Lua脚本

5. 在Redis中运行Lua脚本

6. 在Redis中运行Lua脚本

7. 在Redis中运行Lua脚本

8. 在 Redis 中，Lua 脚本的语法与 Lua 语言本身非常相似，Lua 脚本的语法与 Lua 语言本身非常相似，Lua 脚本的语法与 Lua 语言本身非常相似。

### 20.1.1 使用 Lua 脚本

在 Redis 中，使用 Lua 脚本的 C API 函数 `lua_open` 来创建 Lua 脚本。在 Redis 中，使用 Lua 脚本的 C API 函数 `lua_open` 来创建 Lua 脚本。在 Redis 中，使用 Lua 脚本的 C API 函数 `lua_open` 来创建 Lua 脚本。

### 20.1.2 脚本库

Redis 提供了 Lua 脚本的库，这些库在 Redis 中是预定义的。Redis 提供了 Lua 脚本的库，这些库在 Redis 中是预定义的。Redis 提供了 Lua 脚本的库，这些库在 Redis 中是预定义的。

- `base library` 库包含了一些基本的函数，如 `assert`、`error`、`pairs`、`tostring`、`pcall` 等。这些函数在 Redis 中是预定义的。
- `table library` 库包含了一些与表相关的函数，如 `table.concat`、`table.insert`、`table.remove`、`table.sort` 等。这些函数在 Redis 中是预定义的。
- `string library` 库包含了一些与字符串相关的函数，如 `string.find`、`string.format`、`string.len`、`string.reverse` 等。这些函数在 Redis 中是预定义的。

·`math library` `C` `math.abs` `math.max` `math.min` `math.sqrt` `math.log`

·`debug library` `debug.sethook` `debug.gethook` `debug.getinfo` `debug.setmetatable` `debug.getmetatable`

·`Lua CJSON` <http://www.kyne.com.au/~mark/software/lua-cjson.php> `UTF-8` `JSON` `cjson.decode` `JSON` `Lua` `cjson.encode` `Lua` `JSON`

·`Struct` <http://www.inf.puc-rio.br/~roberto/struct/> `Lua` `C` `struct.pack` `Lua` `struct-like` `struct.unpack` `Lua`

·`Lua msgpack` [https://github.com/antirez/lua-](https://github.com/antirez/lua-msgpack) `msgpack` `MessagePack` `msgpack.pack` `Lua` `MessagePack` `msgpack.unpack` `MessagePack` `Lua`

`Lua` `Redis`

### 20.1.3 `redis` 模块

模块名称为 `Lua` 模块名称为 `redis` 模块名称为 `table` 模块名称为 `redis`

· `Redis` 模块 `redis.call` `redis.pcall`

· `Redis` 模块 `log` 模块 `redis.log` 模块名称为 `level` 模块名称为 `redis.LOG_DEBUG` `redis.LOG_VERBOSE` `redis.LOG_NOTICE` `redis.LOG_WARNING`

· `SHA1` 模块 `redis.sha1hex`

· `redis.error_reply` `redis.status_reply`

模块名称为 `redis.call` `redis.pcall` 模块名称为 `Lua` 模块名称为 `Redis`

---

```
redis> EVAL "return redis.call('PING')" 0
PONG
```

---

### 20.1.4 `Redis` 模块名称为 `Lua` 模块名称为

模块名称为 `Redis` 模块名称为 `Lua` 模块名称为 `side effect` 模块名称为 `pure function`



```

--random-with-default-seed.lua
math.randomseed(redis.call('RANDOMSEED'))

```

```

--random-with-default-seed.lua
math.randomseed(redis.call('RANDOMSEED'))

```

```

--random-with-default-seed.lua
math.randomseed(redis.call('RANDOMSEED'))

```

```

--random-with-default-seed.lua
math.randomseed(redis.call('RANDOMSEED'))

```

```

--random-with-default-seed.lua
math.randomseed(redis.call('RANDOMSEED'))

```

---

```

--random-with-default-seed.lua
local i = 10
local seq = {}
while (i > 0) do
 seq[i] = math.random(i)
 i = i-1
end
return seq

```

---

```

--random-with-default-seed.lua

```

---

```

$ redis-cli --eval random-with-default-seed.lua
1) (integer) 1
2) (integer) 2
3) (integer) 2
4) (integer) 3
5) (integer) 4

```

6) (integer) 4  
7) (integer) 7  
8) (integer) 1  
9) (integer) 7  
10) (integer) 2

---

redis-cli --eval math.randomseed seed

10086

---

```
--random-with-new-seed.lua
math.randomseed(10086) --change seed
local i = 10
local seq = {}
while (i > 0) do
 seq[i] = math.random(i)
 i = i-1
end
return seq
```

---

redis-cli --eval seed 0

---

```
$ redis-cli --eval random-with-new-seed.lua
1) (integer) 1
2) (integer) 1
3) (integer) 2
4) (integer) 1
5) (integer) 1
6) (integer) 3
7) (integer) 1
8) (integer) 1
9) (integer) 3
10) (integer) 1
```

---

## 20.1.5 随机数

Redisのmath  
math.random  
math.randomseed

Lua  
Luaの関数

RedisのLua

---

```
redis> SADD fruit apple banana cherry
(integer) 3
redis> SMEMBERS fruit
1) "cherry"
2) "banana"
3) "apple"
redis> SADD another-fruit cherry banana apple
(integer) 3
redis> SMEMBERS another-fruit
1) "apple"
2) "banana"
3) "cherry"
```

---

fruit  
another-fruit  
SMEMBERS

RedisのSMEMBERS  
“”

·SINTER

·SUNION

·SDIFF

·SMEMBERS

·HKEYS

·HVALS

·KEYS

Redis Lua 脚本函数

`__redis__compare_helper` Lua 脚本函数

`__redis__compare_helper` 脚本函数 `table.sort` 脚本函数

Redis Lua 脚本函数

Redis Lua 脚本函数 `fruit` `another-fruit` `SMEMBERS`

Redis Lua 脚本函数 `SMEMBERS` 脚本函数

---

```
redis> EVAL "return redis.call('SMEMBERS', KEYS[1])" 1 fruit
1) "apple"
2) "banana"
3) "cherry"
redis> EVAL "return redis.call('SMEMBERS', KEYS[1])" 1 another-fruit
1) "apple"
2) "banana"
3) "cherry"
```

---

## 20.1.6 Redis.pcall 脚本函数

```
__redis__err_handler Redis
redis.pcall Redis
__redis__err_handler

Lua
```

---

```
--
1
--
2
--
3
return redis.pcall('wrong command')
```

---

---

```
$ redis-cli --eval wrong-command.lua
(error) @user_script: 4: Unknown Redis command called from Lua script
```

---

```
@user_script 4
Lua
```

## 20.1.7 Lua

```
Lua
local Lua
```

Redis Lua 20-1

---

```
redis> EVAL "x = 10" 0
(error) ERR Error running script
(call to f_df1ad3745c2d2f078f0f41377a92bb6f8ac79af0):
@enable_strict_lua:7: user_script:1:
Script attempted to create global variable 'x'
```

---

Redis Lua 20-1

---

```
redis> EVAL "return x" 0
(error) ERR Error running script
(call to f_03c387736bb5cc009ff35151572cee04677aa374):
@enable_strict_lua:14: user_script:1:
Script attempted to access unexisting global variable 'x'
```

---

Redis Lua 20-1

Redis Lua 20-1

---

```
redis> EVAL "redis = 10086; return redis" 0
(integer) 10086
```

---

## 20.1.8 Lua 20-1

Redis Lua 20-1

Lua 20-1

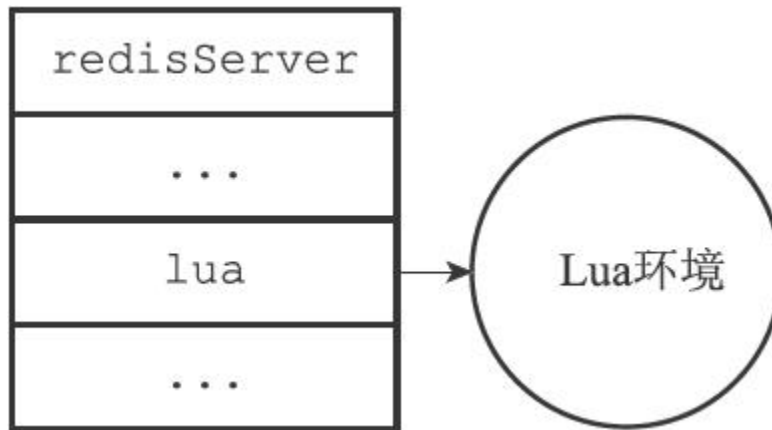


图20-1 集成Lua

Redis 通过 Lua 脚本实现 Redis 命令的扩展，Lua 脚本在 Redis 中运行，Redis 通过 Lua 脚本实现 Redis 命令的扩展。

## 20.2 Lua

Redis 2.8.10 版本开始支持 Lua 脚本。Redis 2.8.10 版本开始支持 Lua 脚本，可以在 Redis 2.8.10 版本开始支持 Lua 脚本。Redis 2.8.10 版本开始支持 Lua 脚本。

Redis 2.8.10 版本开始支持 Lua 脚本。

### 20.2.1 安装

Redis 2.8.10 版本开始支持 Lua 脚本。Redis 2.8.10 版本开始支持 Lua 脚本。Redis 2.8.10 版本开始支持 Lua 脚本。Redis 2.8.10 版本开始支持 Lua 脚本。

Lua 脚本 redis.call redis.pcall Redis 2.8.10 版本开始支持 Lua 脚本。

1. Lua 脚本 redis.call redis.pcall Redis 2.8.10 版本开始支持 Lua 脚本。

2. Redis 2.8.10 版本开始支持 Lua 脚本。

3. Redis 2.8.10 版本开始支持 Lua 脚本。

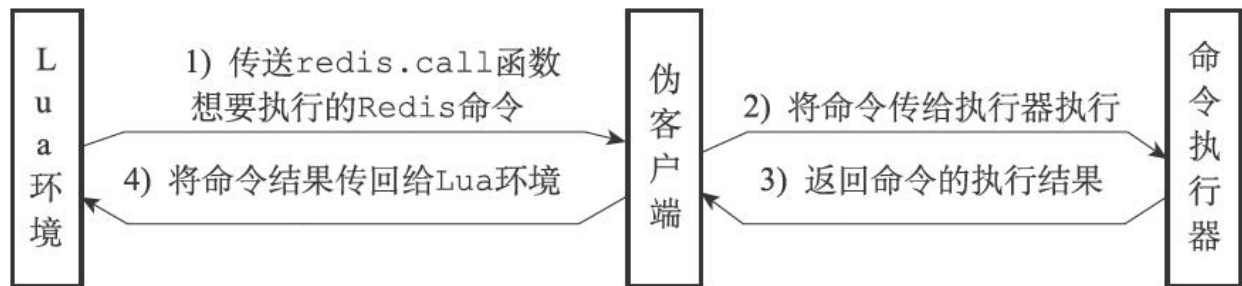
4. Redis 2.8.10 版本开始支持 Lua 脚本。

5. Lua 脚本 redis.call redis.pcall Redis 2.8.10 版本开始支持 Lua 脚本。



6 redis.call redis.pcall

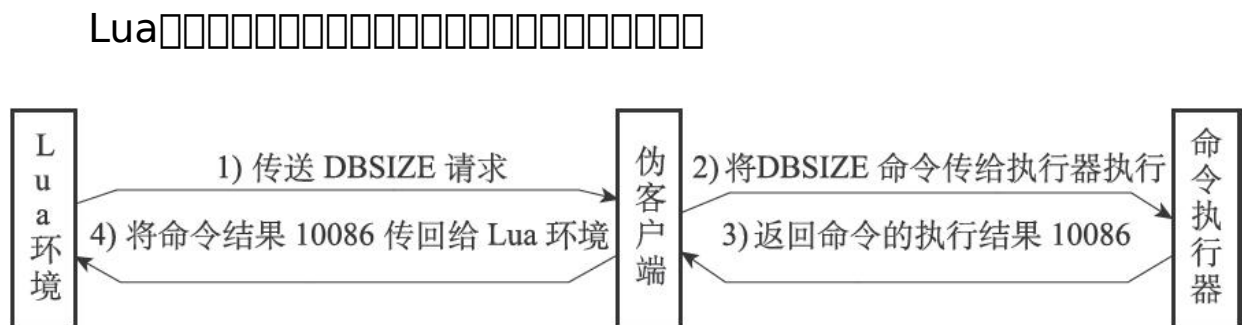
20-2 Lua redis.call Lua redis.pcall



20-2 Lua Redis

20-3 Lua

```
redis> EVAL "return redis.call('DBSIZE')" 0
(integer) 10086
```



20-3 Lua DBSIZE

## 20.2.2 lua\_scripts

Redis Lua `lua_scripts`  
Lua SHA1 checksum SHA1  
Lua

---

```
struct redisServer {
 // ...
 dict *lua_scripts;
 // ...
};
```

---

Redis EVAL Lua SCRIPT LOAD  
Lua `lua_scripts`

---

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
redis> SCRIPT LOAD "return 1+1"
"a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9"
redis> SCRIPT LOAD "return 2*2"
"4475bfb5919b5ad16424cb50f74d4724ae833e72"
```

---

`lua_scripts` SCRIPT LOAD Lua  
20-4



## 20-4 lua\_scripts

lua\_scripts[  
 SCRIPT\_EXISTS  
 lua\_scripts

## 20.3 EVAL

EVAL

1 Lua Lua Lua

2 lua\_scripts

3 Lua Lua

---

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

---

EVAL

### 20.3.1

EVAL Lua Lua

Lua Lua f\_

SHA1 body

---

EVAL "return 'hello world'" 0

---

lua\_scripts Lua

---

```
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91()
 return 'hello world'
end
```

---

return'hello world' SHA1  
5332031c6b470dc5a0dd9b4bf2030dea6d65de91  
f\_5332031c6b470dc5a0dd9b4bf2030dea6d65de91  
return'hello world'

·

·lua

·lua SHA1

lua EVALSHA

EVALSHA

### 20.3.2 lua\_scripts

EVAL lua\_scripts  
lua\_scripts

---

EVAL "return 'hello world'" 0

---

lua\_scripts Lua SHA1  
5332031c6b470dc5a0dd9b4bf2030dea6d65de91

---

Lua

---

return 'hello world'

---

lua\_scripts 20-5

|                                            |                          |
|--------------------------------------------|--------------------------|
| lua_scripts                                |                          |
| ...                                        |                          |
| "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" | → "return 'hello world'" |
| ...                                        |                          |

20-5 lua\_scripts

### 20.3.3

lua\_scripts  
lua\_scripts



3 Lua

f\_5332031c6b470dc5a0dd9b4bf2030dea6d65de91

4

5 f\_5332031c6b470dc5a0dd9b4bf2030dea6d65de91

"hello world"

6 Lua

---

EVAL "return 'hello world'" 0

---

EVAL



## 20.4 EVALSHA□□□□□

```

000000EVAL000000000000EVAL00000000Lua0000Lua000
000
000f_000040000000SHA100000000
0f_5332031c6b470dc5a0dd9b4bf2030dea6d65de910

```

```
luaevalsha1.lua
```

□ □ □ □ □ □ □ □ □ □ □ □ □ □

```
def EVALSHA(sha1):
 #
 #####
 #
 #####f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91
 func_name = "f_" + sha1
 #
 #####Lua
 #####
 if function_exists_in_lua_env(func_name):
 #
 #####
 execute_lua_function(func_name)
 else:
 #
 #####
 send_script_error("SCRIPT NOT FOUND")
```

☐☐☐☐☐☐☐☐☐☐ EVAL☐☐☐☐

---

```
redis> EVAL "return 'hello world'" 0
"hello world"
```

---

Lua

---

```
function f_5332031c6b470dc5a0dd9b4bf2030dea6d65de91()
 return 'hello world'
end
```

---

EVALSHA

---

```
redis> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
"hello world"
```

---

SHA1

f\_5332031c6b470dc5a0dd9b4bf2030dea6d65de91

Lua

f\_5332031c6b470dc5a0dd9b4bf2030dea6d65de91

"hello world"

## 20.5 脚本引擎

Redis 提供了 `EVAL`、`EVALSHA`、`SCRIPT FLUSH`、`SCRIPT EXISTS`、`SCRIPT LOAD`、`SCRIPT KILL` 等命令。

这些命令用于管理 Redis 的 Lua 脚本。

### 20.5.1 SCRIPT FLUSH

`SCRIPT FLUSH` 命令用于清除 Redis 的 Lua 脚本。该命令会清除 `lua_scripts` 字典中的所有脚本。

该命令的语法如下：

---

```
def SCRIPT_FLUSH():
 #
 dictRelease(server.lua_scripts)
 #
 server.lua_scripts = dictCreate(...)
 #
 lua
 lua_close(server.lua)
 #
 lua
 server.lua = init_lua_env()
```

---

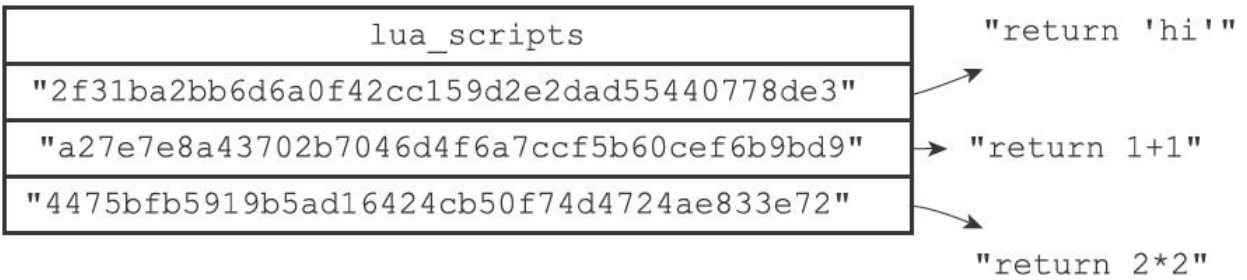
## 20.5.2 SCRIPT EXISTS

**SCRIPT EXISTS**

SCRIPT EXISTS[ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] lua\_scripts [ ] [ ] [ ] [ ]

□ □ □ □ □ □ □ □ □ □

```
def SCRIPT_EXISTS(*sha1_list):
 #
 result_list = []
 #
 SHA1
 for sha1 in sha1_list:
 #
 lua_scripts
 #
 #
 #
 if sha1 in server.lua_scripts:
 #
 1
 result_list.append(1)
 else:
 #
 0
 result_list.append(0)
 #
 send list reply(result_list)
```



## 20-6 lua\_scripts

20-6 lua\_scripts

```
redis> SCRIPT EXISTS "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
1) (integer) 1
redis> SCRIPT EXISTS "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9"
1) (integer) 1
redis> SCRIPT EXISTS "4475bfb5919b5ad16424cb50f74d4724ae833e72"
1) (integer) 1
redis> SCRIPT EXISTS "NotExistsScriptSha1HereABCDEFGHIJKLMNOPQ"
1) (integer) 0
```



SCRIPT EXISTS SHA1 SHA1

SCRIPT EXISTS lua\_scripts lua\_scripts  
 SCRIPT EXISTS Lua SHA1  
 Lua lua\_scripts SHA1

### 20.5.3 SCRIPT LOAD

```
SCRIPT LOAD[]EVAL[]
Lua[]lua_scripts[]
```

[illegible]

```
redis> SCRIPT LOAD "return 'hi'"
"2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
```

□□□□□□Lua□□□□□□□□□□

```
function f_2f31ba2bb6d6a0f42cc159d2e2dad55440778de3()
 return 'hi'
end
```

```

 ""2f31ba2bb6d6a0f42cc159d2e2dad55440778de3"
 "return'hi'"lua_scripts20-7

```

|                                            |
|--------------------------------------------|
| lua_scripts                                |
| ...                                        |
| "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" |
| ...                                        |

→ "return 'hi'"

## □20-7 lua\_scripts□□

```

0000000000000000EVALSHA00000000SCRIPT LOAD000
00000000

```

---

```
redis> EVALSHA "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" 0
"hi"
```

---

## 20.5.4 SCRIPT KILL

lua-time-limit 设置 Lua 脚本执行时间限制，单位为毫秒。当脚本执行时间超过该限制时，Redis 会返回错误信息，并自动终止该脚本。该配置项可以通过 Redis 配置文件或命令行选项进行设置。

lua-time-limit 配置项的默认值为 0，表示不限制脚本执行时间。当该配置项被设置为非 0 值时，Redis 会在脚本执行过程中定期检查其运行时间。一旦超过设定的时间限制，Redis 会立即返回错误信息，并执行 SCRIPT KILL 命令来终止该脚本。在 Redis 2.8.10 版本之前，该配置项被称为 SHUTDOWN 命令。

20-8

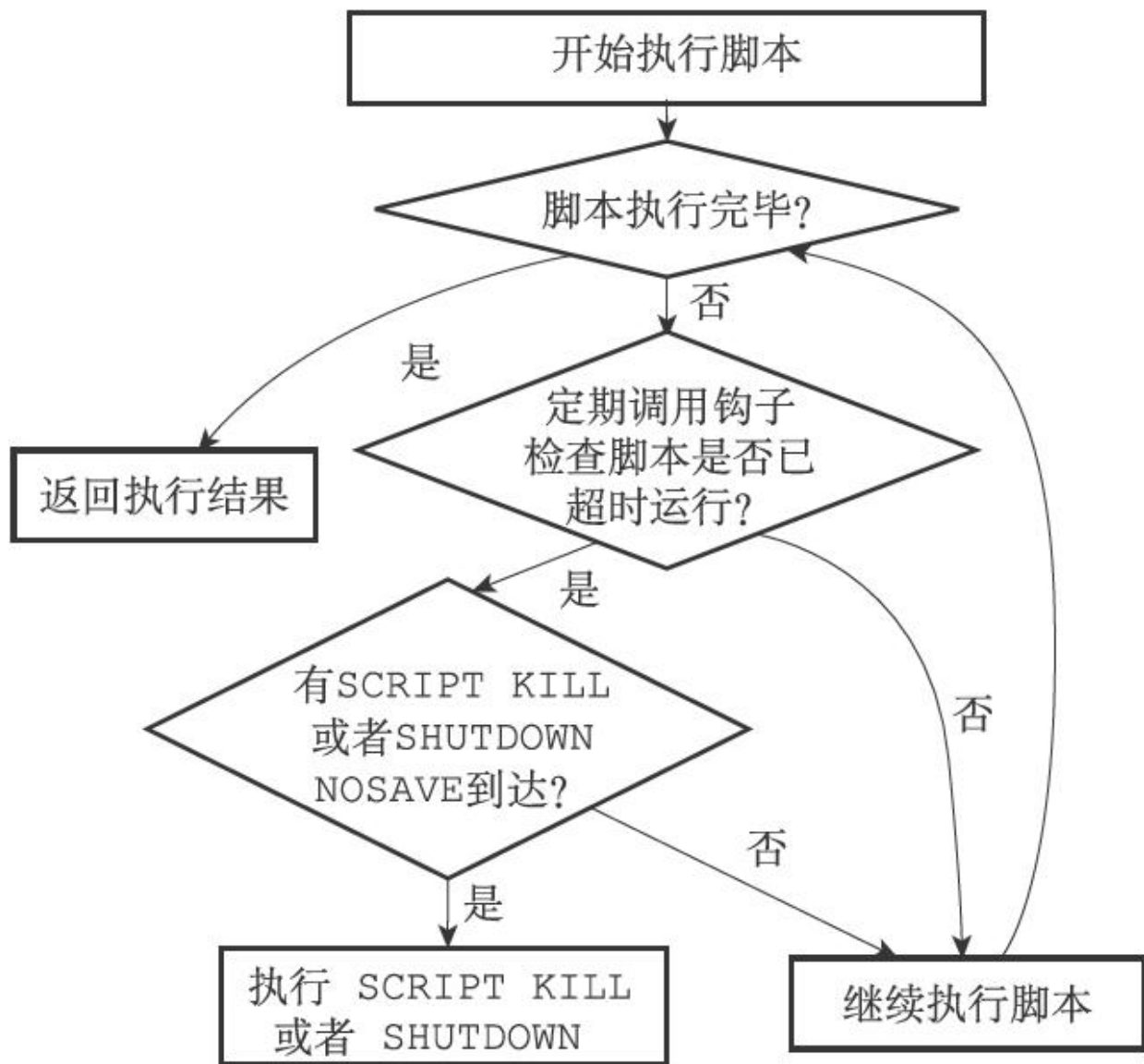


图20-8 脚本执行流程图

脚本执行过程中，如果脚本执行时间超过设定的超时时间，则会调用钩子函数，检查脚本是否已超时运行。如果脚本已超时运行，则会调用钩子函数，检查是否有SCRIPT KILL或者SHUTDOWN NOSAVE到达。如果有SCRIPT KILL或者SHUTDOWN NOSAVE到达，则会执行SCRIPT KILL或者SHUTDOWN NOSAVE。如果没有SCRIPT KILL或者SHUTDOWN NOSAVE到达，则会继续执行脚本。

脚本执行过程中，如果脚本执行时间超过设定的超时时间，则会调用钩子函数，检查脚本是否已超时运行。如果脚本已超时运行，则会调用钩子函数，检查是否有SHUTDOWN nosave到达。如果有SHUTDOWN nosave到达，则会执行SHUTDOWN nosave。如果没有SHUTDOWN nosave到达，则会继续执行脚本。



## 20.6 □□□□

```

Redis
EVAL EVALSHA SCRIPT FLUSH SCRIPT
LOAD

```

[illegible]

### 20.6.1 `EVAL` `SCRIPT FLUSH` `SCRIPT LOAD`

```
Redis EVAL SCRIPT FLUSH SCRIPT LOAD
Redis
propagate 20-9
```

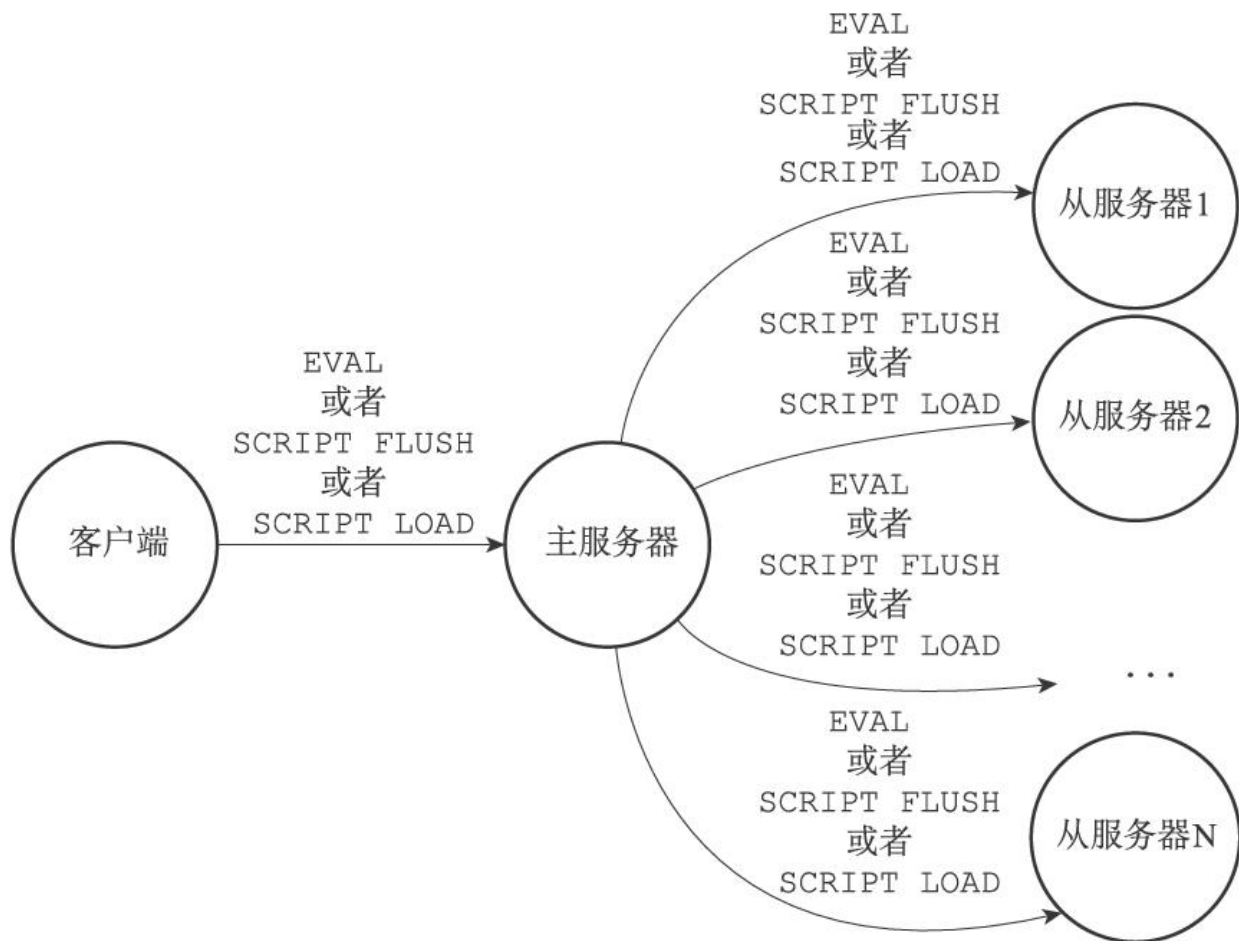


图20-9 Redis命令复制

## 1.EVAL

EVAL命令用于在Redis中执行Lua脚本。

命令格式如下：

```
redis> EVAL "return redis.call('SET', KEYS[1], ARGV[1])" 1 "msg" "hello world"
OK
```

redis> EVAL "return redis.call('SET', KEYS[1], ARGV[1])" 0 "msg" "hello world"

---

```
"return redis.call('SET', KEYS[1], ARGV[1])"
```

---

redis>

## 2.SCRIPT FLUSH

redis> SCRIPT FLUSH

redis>

## 3.SCRIPT LOAD

redis> SCRIPT LOAD "return 'hello world'"

redis>

---

```
redis> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

---

redis>

---

---

```
SCRIPT LOAD "return 'hello world'"
```

---

```
00000000000000000000000000000000
```

---

```
"return 'hello world'"
```

---

## 20.6.2 EVALSHA

EVALSHA 命令使用 Lua 脚本在 Redis 服务器内部执行。EVALSHA 命令与 EVAL 命令类似，但 EVALSHA 命令使用脚本的 SHA1 值来标识脚本。EVALSHA 命令的语法如下：

```
EVALSHA <script-sha1> <numkeys> <key1> <key2> ... <keyN>
```

其中，<script-sha1> 是脚本的 SHA1 值，<numkeys> 是脚本中使用的键的数量，<key1> 到 <keyN> 是脚本中使用的键名。EVALSHA 命令返回脚本的执行结果。如果脚本执行成功，则返回结果；如果脚本执行失败，则返回 not found。

```
00000000000000000000000000000000master000000000000000000000000
```

---

```
master> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

---

```
00000000SCRIPT LOAD000000SHA100
5332031c6b470dc5a0dd9b4bf2030dea6d65de910000000000
000000
```

```
000000000000slave10000000000master0000master0000000000
```

---

```
"return 'hello world'"
```

---

```
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 slave1
```

---

```
master> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
"hello world"
```

---

```
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 master
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 slave1
```

---

```
slave1> EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
(error) NOSCRIPT No matching script. Please use EVAL.
```

---

```
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 Lua
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 EVALSHA
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 EVALSHA
```

---

```
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 master
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 slave1
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 master
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 Lua
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 slave1
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 EVAL
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 SCRIPT LOAD
```

---

```
redis-cli -h 127.0.0.1 -p 6379 -a 1234567890 master
```

---

```
master> SCRIPT LOAD "return 'hello world'"
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"
```

---

redis-cli -h 127.0.0.1 -p 6379 slave1 --master slave1 --evalsha 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 Lua

redis-cli -h 127.0.0.1 -p 6379 slave2 --master master --master

---

```
"return 'hello world'"
```

---

```
redis-cli -h 127.0.0.1 -p 6379 slave2 --master master --evalsha 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 0 "hello world"
```

---

redis-cli -h 127.0.0.1 -p 6379 master --slave1 127.0.0.1 --evalsha 5332031c6b470dc5a0dd9b4bf2030dea6d65de91

redis-cli -h 127.0.0.1 -p 6379 Redis --evalsha 5332031c6b470dc5a0dd9b4bf2030dea6d65de91 EVALSHA EVALSHA EVAL EVAL EVALSHA

redis-cli -h 127.0.0.1 -p 6379 EVALSHA EVALSHA EVAL lua\_scripts repl\_scriptcache\_dict Redis EVALSHA

## 1. EVALSHA

repl\_scriptcache\_dict

```
struct redisServer {
 // ...
 dict *repl_scriptcache_dict;
 // ...
};
```

repl\_scriptcache\_dict

Lua

SHA1

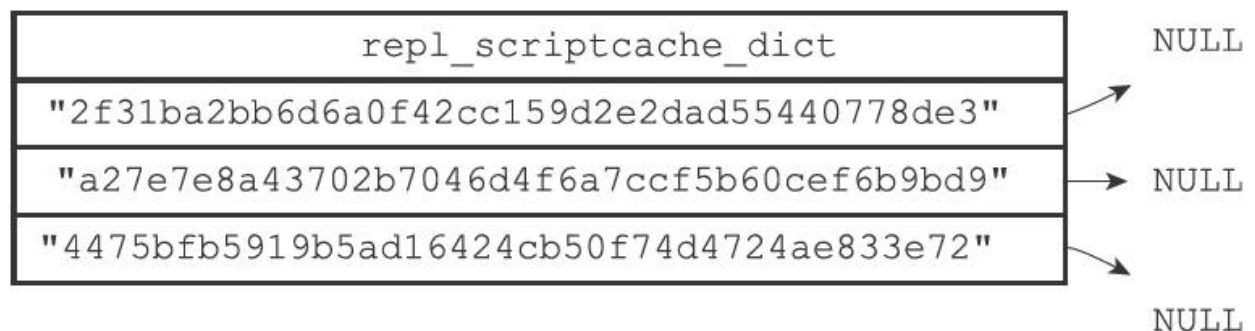
NULL

repl\_scriptcache\_dict

Lua

SHA1

EVALSHA



20-10 repl\_scriptcache\_dict

repl\_scriptcache\_dict

20-10

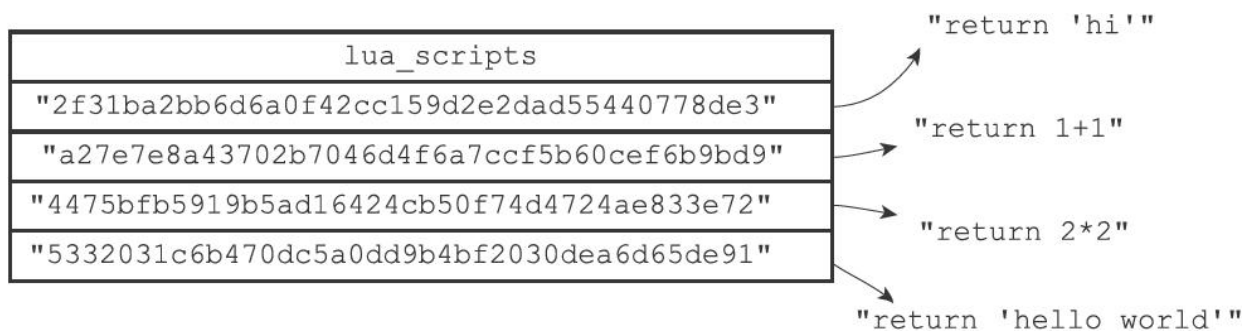
EVALSHA

EVALSHA

```
EVALSHA "2f31ba2bb6d6a0f42cc159d2e2dad55440778de3" ...
EVALSHA "a27e7e8a43702b7046d4f6a7ccf5b60cef6b9bd9" ...
```

EVALSHA "4475bfb5919b5ad16424cb50f74d4724ae833e72" ...

lua\_scripts  
repl\_scriptcache\_dict  
SHA1  
EVALSHA



## 20-11 lua\_scripts

lua\_scripts  
repl\_scriptcache\_dict  
SHA1

"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"

"return 'hello world'"

lua\_scripts  
repl\_scriptcache\_dict  
"5332031c6b470dc5a0dd9b4bf2030dea6d65de91"



---

```
"return 'hello world'"
```

---

```
redis> EVALSHA "return 'hello world'" 0
```

---

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" ...
```

---

```
redis> EVALSHA "return 'hello world'" 0
```

## 2. repl\_scriptcache\_dict

repl\_scriptcache\_dict 是一个字典，用于存储已经编译好的 Lua 脚本。当 Redis 收到一个 EVAL 命令时，它会先检查这个字典，看看有没有已经编译好的脚本。如果有，就直接执行；如果没有，就编译并存储到字典中，然后再执行。

## 3. EVALSHA 和 EVAL 的区别

EVALSHA 和 EVAL 都是用来执行 Lua 脚本的命令。EVALSHA 需要提供一个 SHA1 哈希值，而 EVAL 则不需要。EVALSHA 的优点是它可以在字典中找到已经编译好的脚本，从而避免重复编译，提高效率。

---

```
EVALSHA <sha1> <numkeys> [key ...] [arg ...]
```

---

```
redis> EVAL "return 'hello world'" 0
```

---

```
EVAL <script> <numkeys> [key ...] [arg ...]
```

---

lua\_scripts

1. lua\_scripts[sha1(lua\_scripts)] = lua\_scripts[sha1(lua\_scripts)]

2. lua\_scripts[EVALSHA(lua\_scripts[EVALSHA(lua\_scripts, sha1(lua\_scripts))], numkeys, key, arg)] = lua\_scripts[EVALSHA(lua\_scripts[EVALSHA(lua\_scripts, sha1(lua\_scripts))], numkeys, key, arg)]

lua\_scripts[20-11] = lua\_scripts[20-10]  
repl\_scriptcache\_dict[sha1(lua\_scripts)] = lua\_scripts[sha1(lua\_scripts)]

---

EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0

---

lua\_scripts

---

EVAL "return 'hello world'" 0

---

lua\_scripts

---

"return 'hello world'"

---

lua\_scripts

lua\_scripts["5332031c6b470dc5a0dd9b4bf2030dea6d65de91"] = lua\_scripts["5332031c6b470dc5a0dd9b4bf2030dea6d65de91"]

lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)

lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)

#### 4. lua\_evalsha1

lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)

1. lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)

2. lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)  
lua\_evalsha1(lua\_State \*L, const char \*sha1, const char \*script, int flags)

20-12

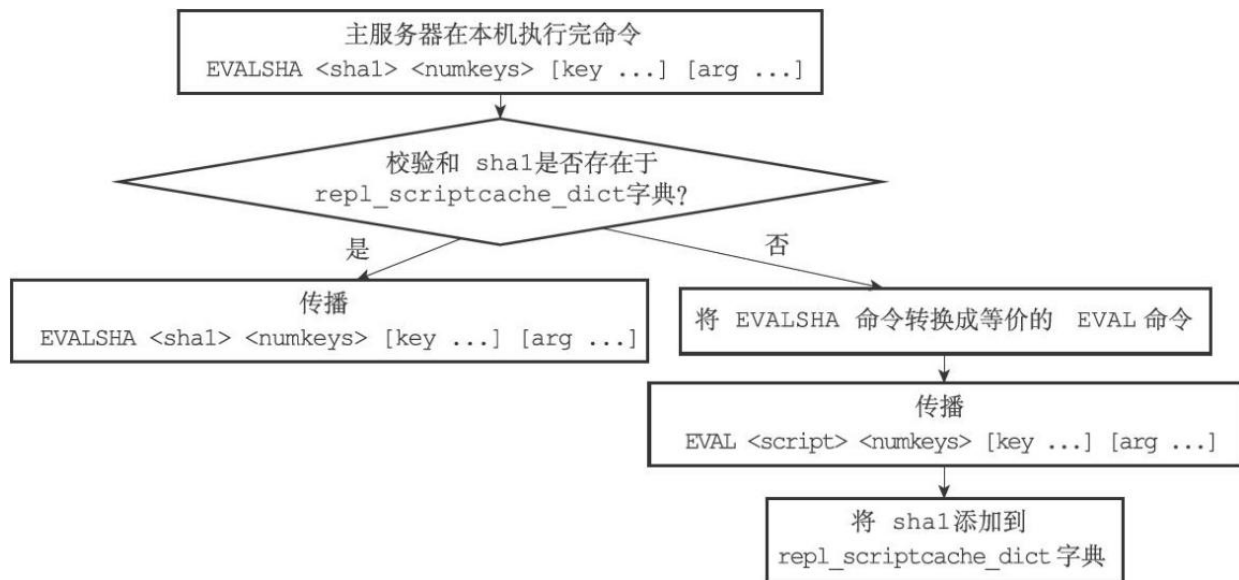


图20-12 EVALSHA命令传播逻辑

主服务器在本地执行完命令后，将命令传播给从服务器。

图20-13 EVALSHA命令传播逻辑

---

```
EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0
```

---

主服务器在本地执行完命令后，将命令传播给从服务器。

---

```
EVAL "return 'hello world'" 0
```

---

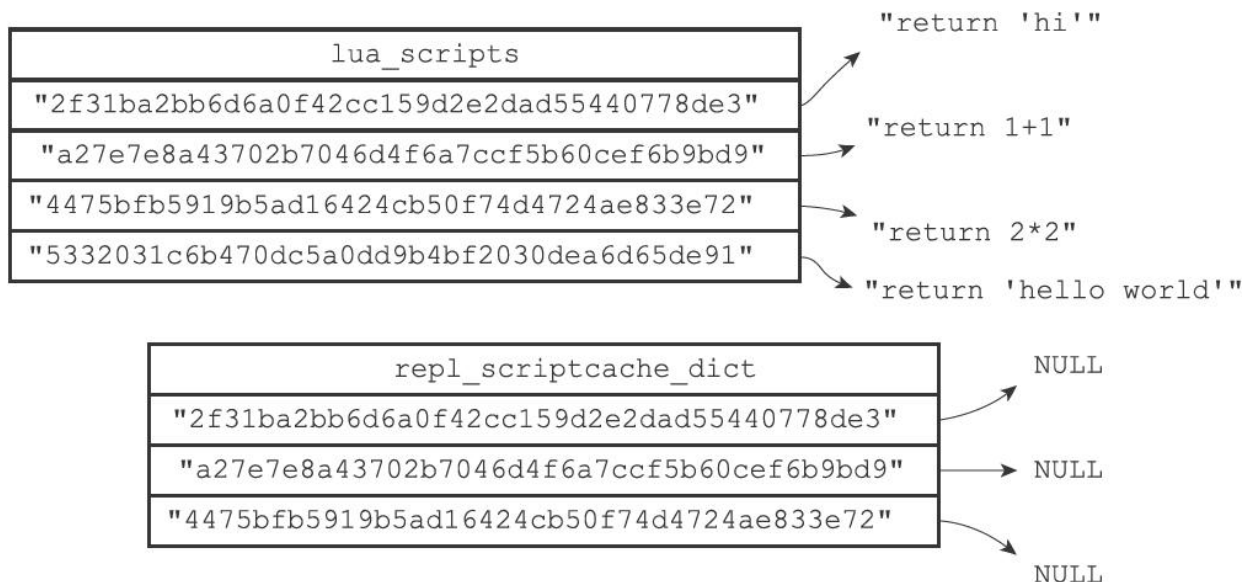


图20-13 EVALSHA 命令使用 lua\_scripts 和 repl\_scriptcache\_dict

Redis 使用 EVAL 命令

Redis 使用 SHA1 命令

Redis 使用 EVALSHA 命令

---

EVALSHA "5332031c6b470dc5a0dd9b4bf2030dea6d65de91" 0

---

Redis 使用 EVALSHA 命令

Redis 使用 EVALSHA 命令

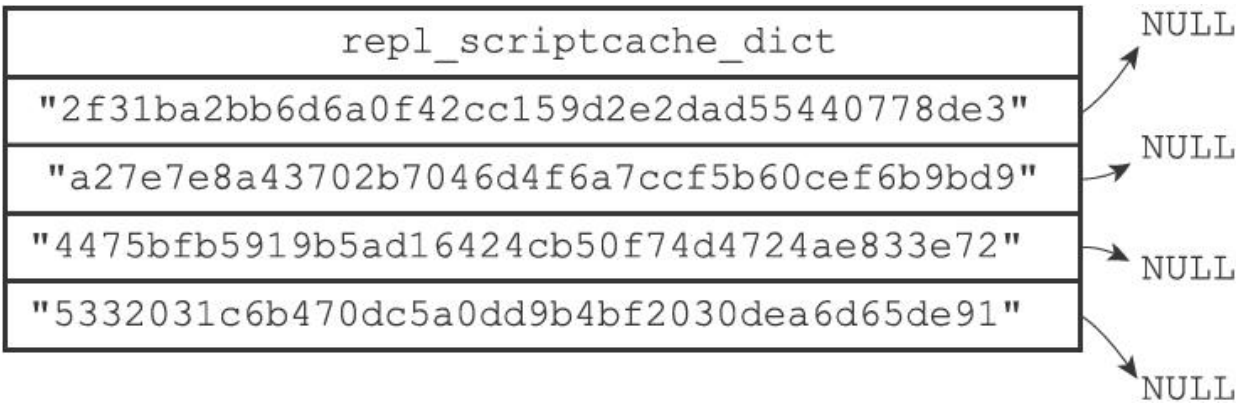


图20-14 EVALSHA操作访问repl\_scriptcache\_dict

## 20.7 Lua

·RedisはLuaを実行可能な環境を提供する。RedisはLuaを実行可能な環境を提供する。

·RedisはLuaを実行可能な環境を提供する。RedisはLuaを実行可能な環境を提供する。

·RedisはLuaを実行可能な環境を提供する。EVAL、SCRIPT LOAD、SCRIPT EXISTSなどのコマンドを実行する。

·EVALはLuaを実行可能な環境を提供する。EVALはLuaを実行可能な環境を提供する。

·EVALSHAはLuaを実行可能な環境を提供する。EVALSHAはLuaを実行可能な環境を提供する。

·SCRIPT FLUSHはlua\_scriptsをクリアする。SCRIPT FLUSHはlua\_scriptsをクリアする。

·SCRIPT EXISTSはSHA1を比較する。SCRIPT EXISTSはSHA1を比較する。

·SCRIPT LOADはLuaを実行可能な環境を提供する。SCRIPT LOADはLuaを実行可能な環境を提供する。

·SCRIPT KILLはLuaを実行可能な環境を提供する。SCRIPT KILLはLuaを実行可能な環境を提供する。

nosave

· EVAL SCRIPT FLUSH SCRIPT LOAD

Redis

· EVALSHA EVALSHA

SHA1 Lua EVALSHA

EVAL EVAL



## 20.8 □□□□

□Lua 5.1 Reference Manual□□Lua□□□□□□□□□□□□□□□□

<http://www.lua.org/manual/5.1/manual.html>

## 21 练习

Redis的SORT命令

对列表进行排序

---

```
redis> RPUSH numbers 5 3 1 4 2
(integer) 5
```

```
#
```

```
查看列表
```

```
redis> LRANGE numbers 0 -1
```

```
1) "5"
```

```
2) "3"
```

```
3) "1"
```

```
4) "4"
```

```
5) "2"
```

```
#
```

```
对列表进行排序
```

```
redis> SORT numbers
```

```
1) "1"
```

```
2) "2"
```

```
3) "3"
```

```
4) "4"
```

```
5) "5"
```

---

对集合进行排序

---

```
redis> SADD alphabet a b c d e f g
(integer) 7
```

```
#
```

```
查看集合
```

```
redis> SMEMBERS alphabet
```

```
1) "d"
```

```
2) "a"
```

```
3) "f"
```

```
4) "e"
```

```
5) "b"
6) "g"
7) "c"
#
redis> SORT alphabet ALPHA
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
7) "g"
```

---

```
redis> SORT test-result BY *weight
1) "jack"
2) "peter"
3) "tom"
#
redis> MSET test-result 3.0 jack 3.5 peter 4.0 tom
(integer) 3
#
redis> ZRANGE test-result 0 -1
1) "jack"
2) "peter"
3) "tom"
#
redis> MSET peter_number 1 tom_number 2 jack_number 3
OK
#
redis> SORT test-result BY *_number
1) "peter"
2) "tom"
3) "jack"
```

---

SORT
 ASC
 DESC
 ALPHA
 LIMIT
 STORE
 BY
 GET
 SORT

SORT

## 21.1 SORT<key>命令

SORT命令

---

SORT <key>

---

对存储在<key>中的元素进行排序

命令格式: SORT <key> [BY <pattern>] [ASC|DESC]

---

```
redis> RPUSH numbers 3 1 2
(integer) 3
redis> SORT numbers
1) "1"
2) "2"
3) "3"
```

---

对numbers列表进行排序

1) 对numbers列表进行排序

[redis.h/redisSortObject](https://redis.h/redisSortObject) 21-1

| array                       |     |
|-----------------------------|-----|
| array[0]<br>redisSortObject | obj |
|                             | u   |
| array[1]<br>redisSortObject | obj |
|                             | u   |
| array[2]<br>redisSortObject | obj |
|                             | u   |

图21-1 numbers数组结构

2. 遍历numbers数组，取出每个obj，取出obj中的u，遍历u中的score，如图21-2所示。

3. 取出obj中的double，遍历double中的u，取出u.score，如图21-3所示。

4. 取出u.score，遍历u.score中的obj，取出obj中的u，遍历u.score，如图21-4所示。

5. 遍历obj中的u，取出u.score，遍历u.score中的obj，取出obj中的u，遍历u.score，如图21-5所示。

redisSortObject <key> redisSortObject SORT numbers redisSortObject

redisSortObject

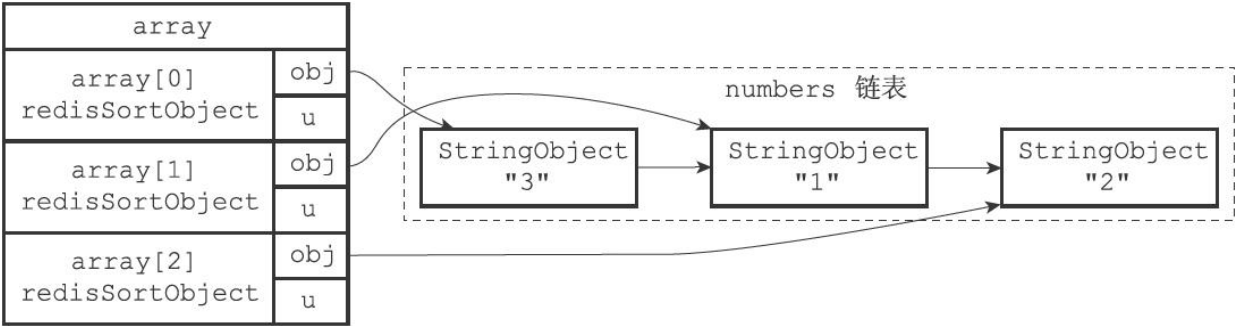


图21-2 redisSortObject的初始状态

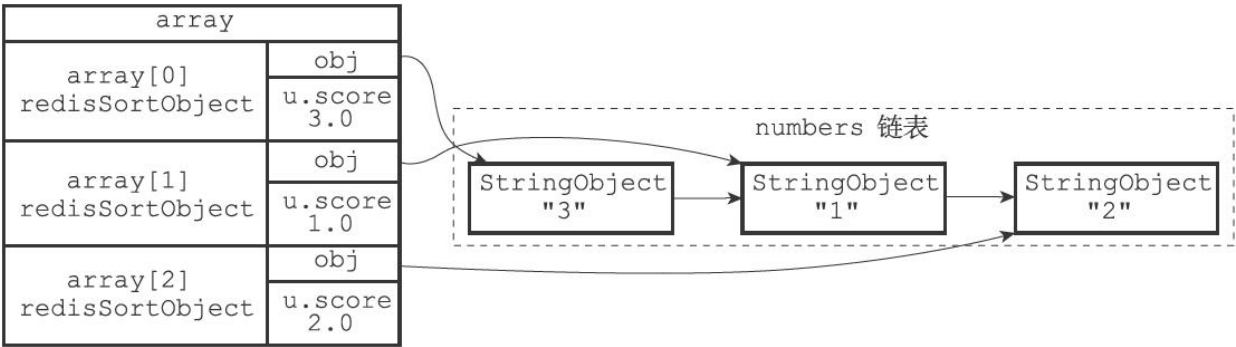


图21-3 为redisSortObject的u.score赋值

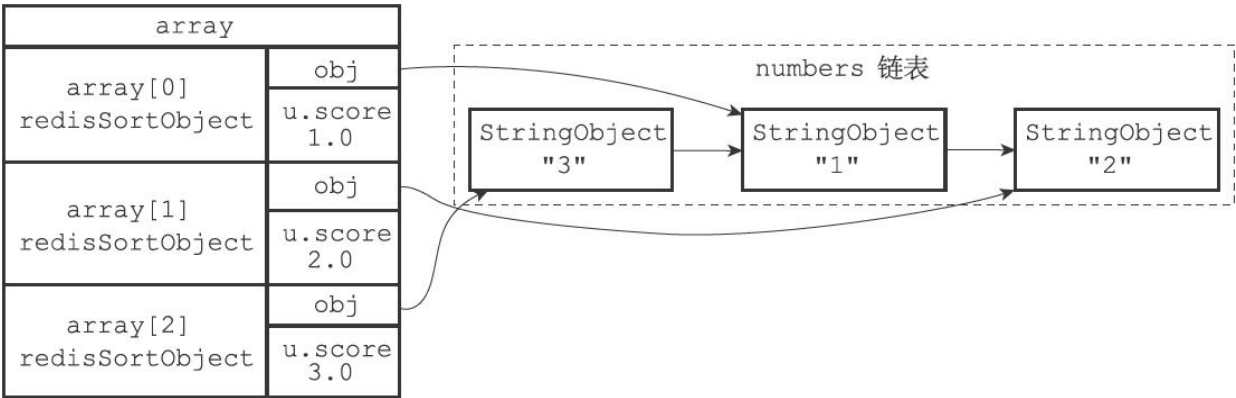


图21-4 继续排序

redisSortObject redisSortObject

---

```
typedef struct _redisSortObject {
 //
 对象
 robj *obj;
 //
 分
 union {
 //
 分数
 double score;
 //
 按BY
 对象
 robj *cmpobj;
 } u;
} redisSortObject;
```

---

**SORT** 命令的语法

redisSortObject 对象是 SORT 命令的输入，redisSortObject 对象

是 SORT 命令的输出



## 21.2 ALPHA

ALPHA SORT

---

`SORT <key> ALPHA`

---

SORT

---

```
redis> SADD fruits apple banana cherry
(integer) 3
```

```
#
```

```
redis> SMEMBERS fruits
```

```
1) "apple"
```

```
2) "cherry"
```

```
3) "banana"
```

```
#
```

```
fruits
```

```
redis> SORT fruits ALPHA
```

```
1) "apple"
```

```
2) "banana"
```

```
3) "cherry"
```

---

SORT fruits ALPHA

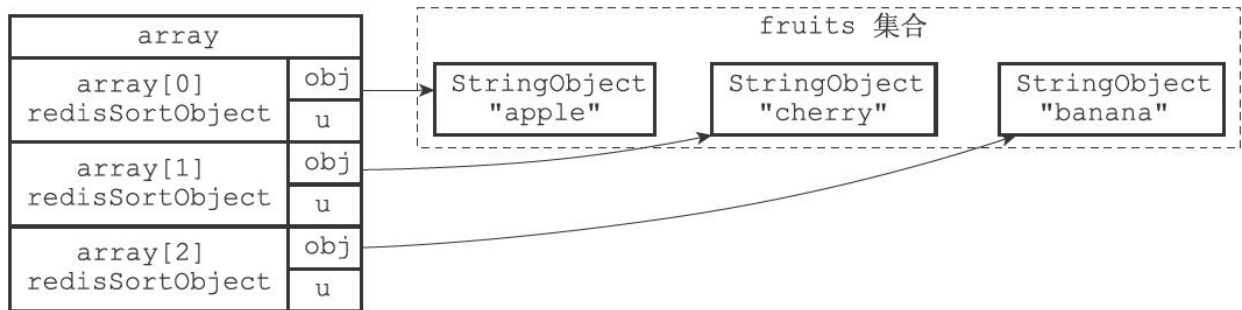
1 redisSortObject fruits

2 obj fruits 21-5

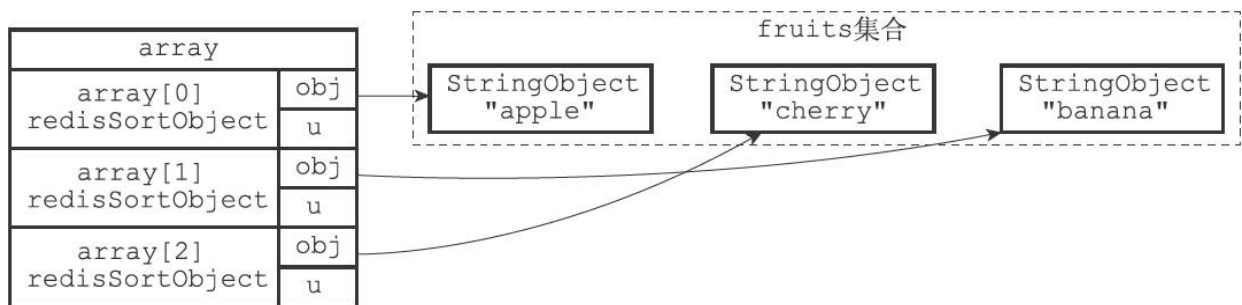
3 obj  
 "apple" "banana" "cherry" "apple"  
 <"banana"<"cherry" "apple"  
 "banana" "cherry" 21-6

4 obj

SORT <key> ALPHA SORT fruits ALPHA



21-5 obj



21-6

## 21.3 ASCDESC

`SORT` 命令可以对列表中的元素进行排序。

语法

---

```
SORT <key>
SORT <key> ASC
```

---

`SORT` 命令可以对列表中的元素进行降序排序。

语法

---

```
SORT <key> DESC
```

---

`numbers` 列表中的元素。

对 `numbers` 列表中的元素进行升序排序。

语法

---

```
redis> RPUSH numbers 3 1 2
(integer) 3
redis> SORT numbers
1) "1"
2) "2"
3) "3"
redis> SORT numbers ASC
1) "1"
2) "2"
3) "3"
```

---

numbers

```
redis> SORT numbers DESC
```

```
1) "3"
2) "2"
3) "1"
```

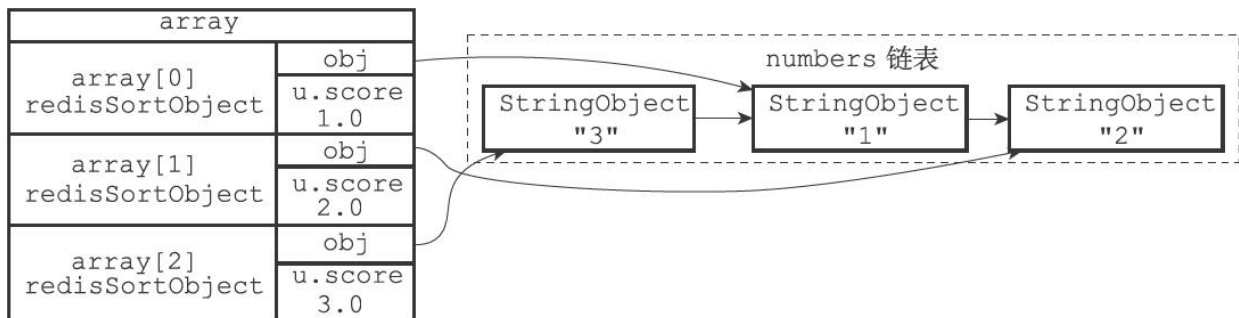
·

·

numbers

·21-7 SORT numbers

·21-8 SORT numbers



21-7

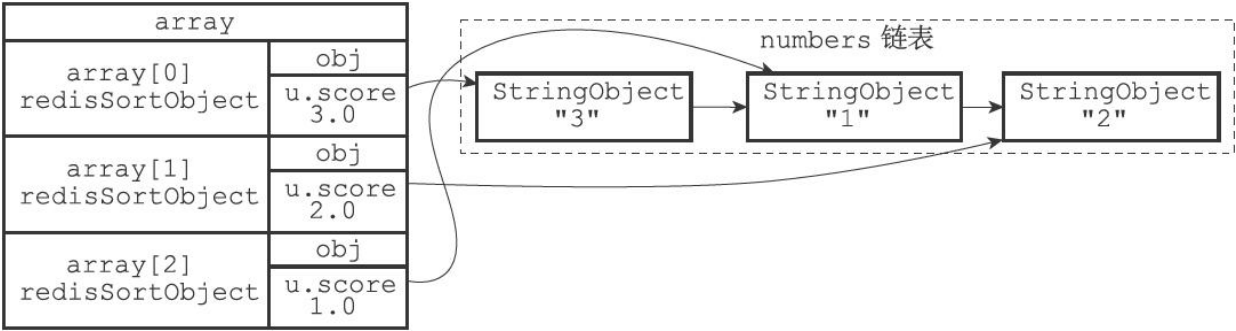


图21-8 内部结构

命令: `sort <Key> DESC` 对集合中的元素进行排序

## 21.4 BY

Redis 的 `SORT` 命令可以对集合中的元素进行排序。默认情况下，元素是按照字典序进行排序的。

我们使用 `fruits` 集合来演示。集合中包含以下元素：

```
"apple" "banana" "cherry"
```

---

```
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> SORT fruits ALPHA
1) "apple"
2) "banana"
3) "cherry"
```

---

Redis 的 `BY` 选项可以用于指定排序的依据。例如，我们可以按照元素的值进行排序。

我们使用 `fruits` 集合来演示。集合中包含以下元素：

---

```
redis> MSET apple-price 8 banana-price 5.5 cherry-price 7
OK
redis> SORT fruits BY *-price
1) "banana"
2) "cherry"
3) "apple"
```

---

Redis 的 `SORT` 命令还可以按照元素的值进行排序。

1 redisSortObject fruits

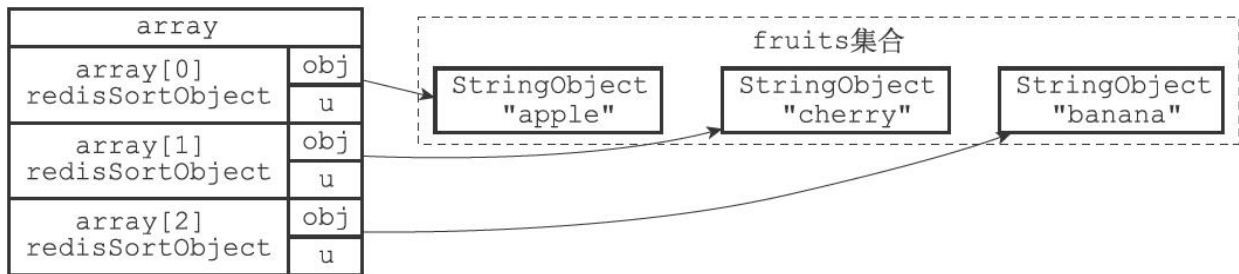
2 obj fruits 21-9

3 obj BY \*-price

· "apple" "apple-price"

· "banana" "banana-price"

· "cherry" "cherry-price"



21-9 obj

4 double u.score

21-10

· "apple" "apple-price" 8.0

· "banana" "banana-price" 5.5

· "cherry" "cherry-price" 7.0

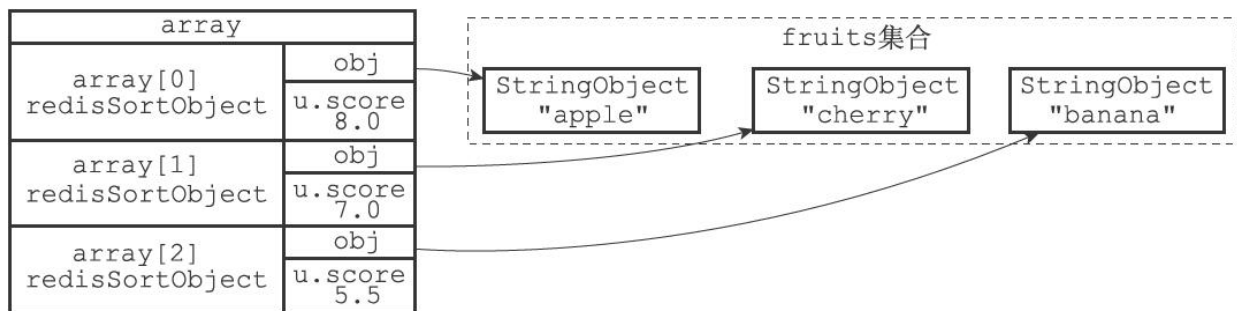


図21-10 並び順をu.scoreで

5番目のu.scoreから並び順をu.scoreで

図21-11

・5.5番"banana"から並び順0番

・7.0番"cherry"から並び順1番

・8.0番"apple"から並び順2番

6番目のobjから並び順

図21-11 SORT<key>BY<pattern>で並び順

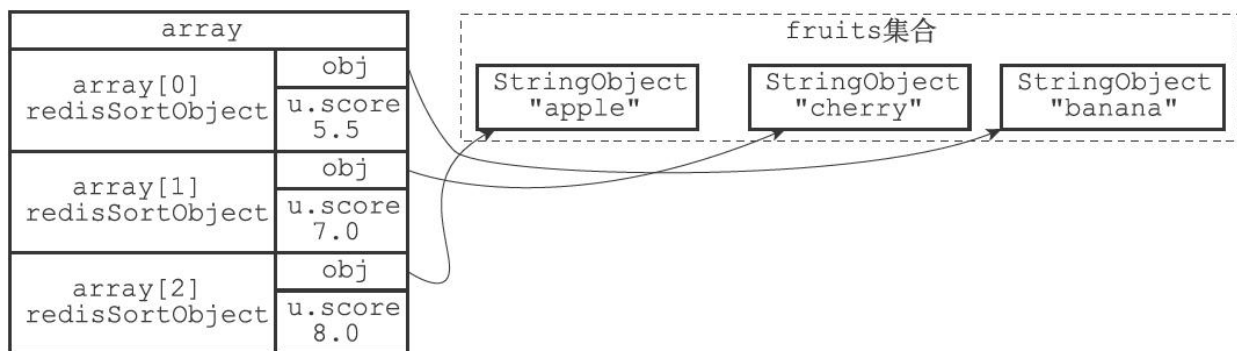


図21-11 u.scoreで並び順



## 21.5 ALPHA와 BY

BY를 사용하면 정렬 기준을 정할 수 있다. 예를 들어, BY를 사용하여 정렬을 하거나 ALPHA를 사용하여 알파벳 순서로 정렬할 수 있다.

예를 들어, fruits라는 키에 다음과 같은 값을 저장한다.

---

```
redis> SADD fruits "apple" "banana" "cherry"
(integer) 3
redis> MSET apple-id "FRUIT-25" banana-id "FRUIT-79" cherry-id "FRUIT-13"
OK
```

---

이제 fruits 키를 정렬해 보자.

---

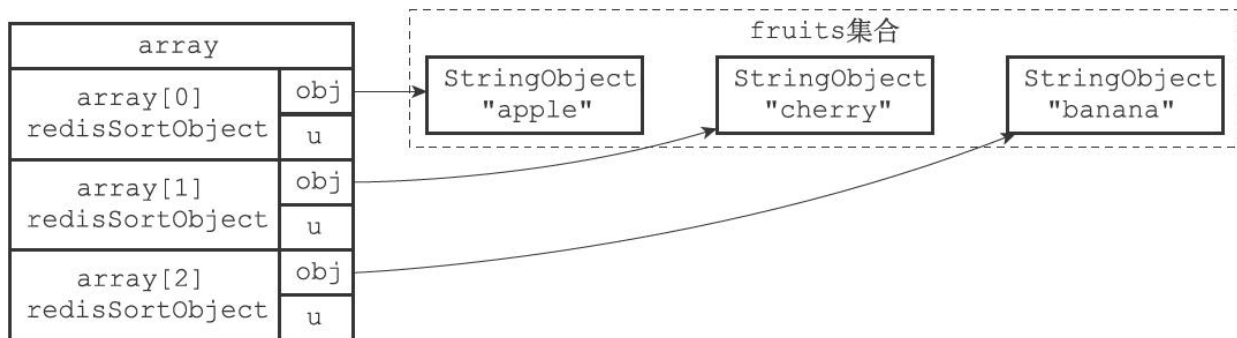
```
redis> SORT fruits BY *-id ALPHA
1)"cherry"
2)"apple"
3)"banana"
```

---

이제 SORT fruits BY \*-id ALPHA 명령어를 실행해 보자.

1. redisSortObject를 사용하여 fruits 키를 정렬한다.

2. 정렬된 객체를 obj라는 변수에 저장한다. 21-12 참조



21-12 obj

3 obj BY \*-id

· "apple" "apple-id"

· "banana" "banana-id"

· "cherry" "cherry-id"

4 u.cmpobj 21-13

5 12-14

· "FRUIT-13" "cherry" 0

· "FRUIT-25" "apple" 1

· "FRUIT-79" "banana" 2



## 21.6 LIMIT

Redis 2.8.10 新增的 LIMIT 命令，用于对集合中的元素进行排序。

---

```
redis> SADD alphabet a b c d e f
(integer) 6
#
集合中的元素
redis> SMEMBERS alphabet
1) "d"
2) "c"
3) "a"
4) "b"
5) "f"
6) "e"
#
集合中的元素
redis> SORT alphabet ALPHA
1) "a"
2) "b"
3) "c"
4) "d"
5) "e"
6) "f"
```

---

Redis 2.8.10 新增的 LIMIT 命令，用于对集合中的元素进行排序。

LIMIT 命令的语法如下：

·offset 指定从集合中的哪个位置开始返回元素。

·count 指定返回多少个元素。

redis> SORT alphabet ALPHA LIMIT 0 4

redis

```
redis> SORT alphabet ALPHA LIMIT 0 4
1) "a"
2) "b"
3) "c"
4) "d"
```

redis> SORT alphabet ALPHA LIMIT 2 3

redis

```
redis> SORT alphabet ALPHA LIMIT 2 3
1) "c"
2) "d"
3) "e"
```

redis> SORT alphabet ALPHA LIMIT 0 4

1) redisSortObject

2) obj alphabet 21-15

3) obj 21-16

4) LIMIT 0 4 0 array[0]

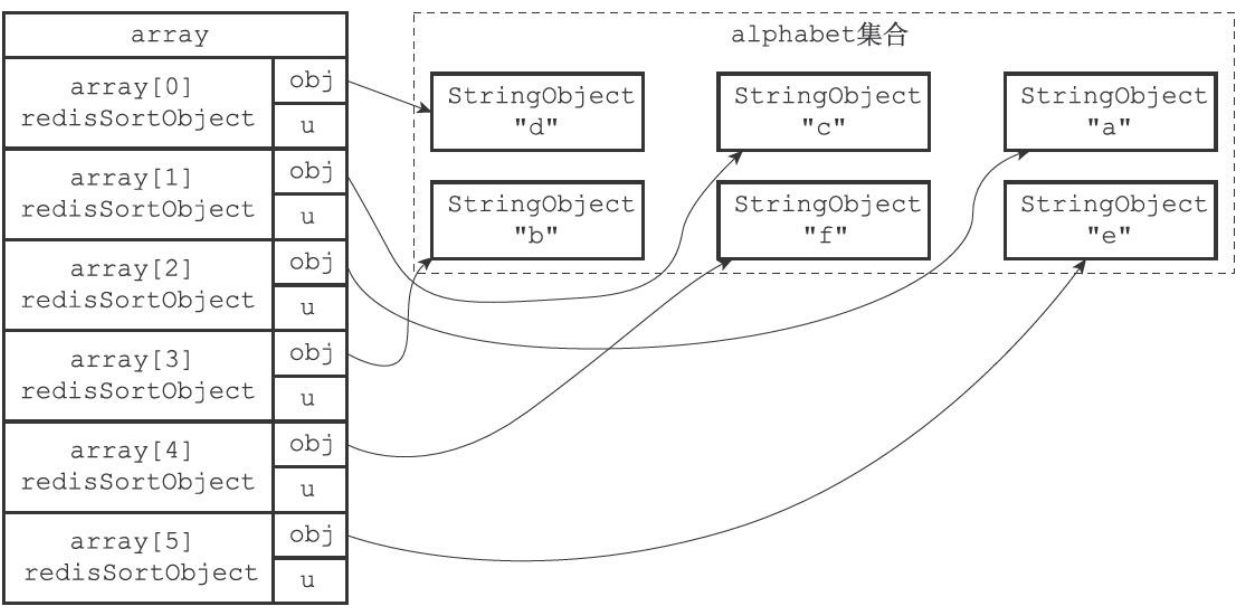
array[1] array[2] array[3] 4 obj

"a" "b" "c" "d"

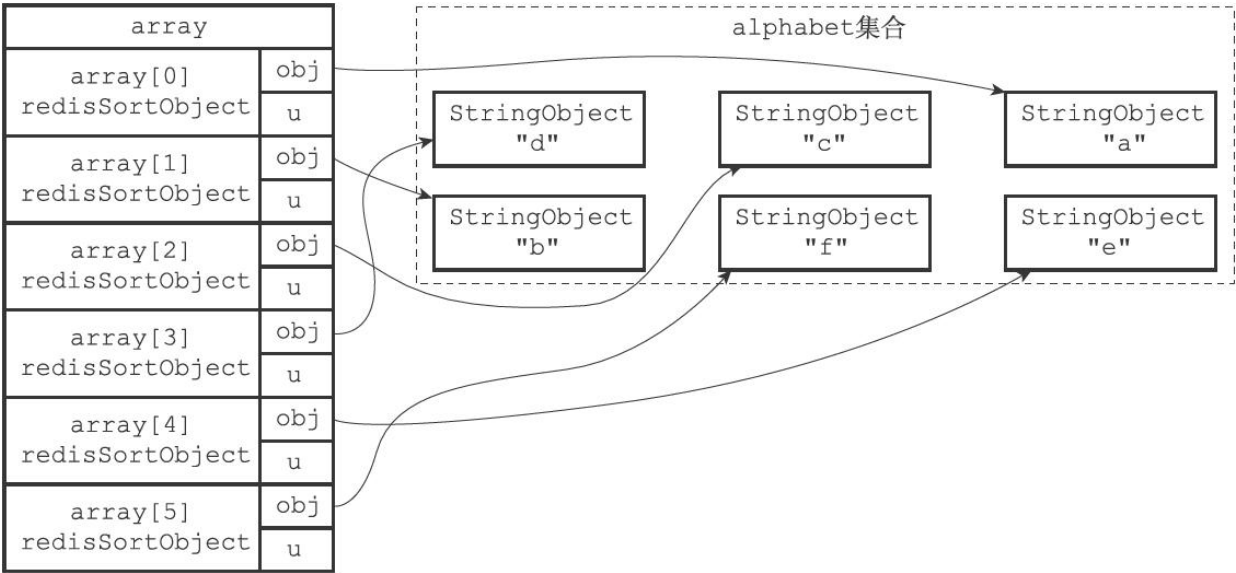
SORT alphabet ALPHA LIMIT 2 3  
 SORT alphabet ALPHA LIMIT 0 4  
 0000

4  
 LIMIT 2 3  
 array[2]  
 array[3] array[4] 3  
 obj "c" "d" "e"  
 0000

SORT LIMIT



21-15 obj



21-16

## 21.7 GET

Redis 提供了 SORT 命令，可以对列表中的元素进行排序。

例如，我们有一个名为 students 的列表，其中包含三个元素，我们使用 SORT 命令对其进行排序。

students 列表

---

```
redis> SADD students "peter" "jack" "tom"
(integer) 3
redis> SORT students ALPHA
1) "jack"
2) "peter"
3) "tom"
```

---

Redis 提供了 GET 命令，可以对列表中的元素进行排序。

GET 命令

Redis 提供了 SORT 命令，可以对列表中的元素进行排序。

Redis 提供了 GET 命令，可以对列表中的元素进行排序。

---

```
#
peter
jack
tom
redis> SET peter-name "Peter White"
OK
redis> SET jack-name "Jack Snow"
OK
redis> SET tom-name "Tom Smith"
OK
```



```
SORT
students
1) "jack"
2) "peter"
3) "tom"
#
jack-name
peter-name
tom-name
redis> SORT students ALPHA GET *-name
1) "Jack Snow"
2) "Peter White"
3) "Tom Smith"
```

---

1. 실행: `SORT students ALPHA GET *-name`

2. 결과: `redisSortObject` 반환

3. 객체: `obj` 반환

4. 배열: `obj` 반환

5. 첫 번째 요소: `0` 번째 요소

6. 두 번째 요소: `1` 번째 요소

7. 세 번째 요소: `2` 번째 요소

8. 실행: `obj` 반환

9. 결과: `GET` 반환

10. 첫 번째 요소: `jack` 반환

·`key="peter" value={"*-name":"peter-name"}`

·`key="tom" value={"*-name":"tom-name"}`

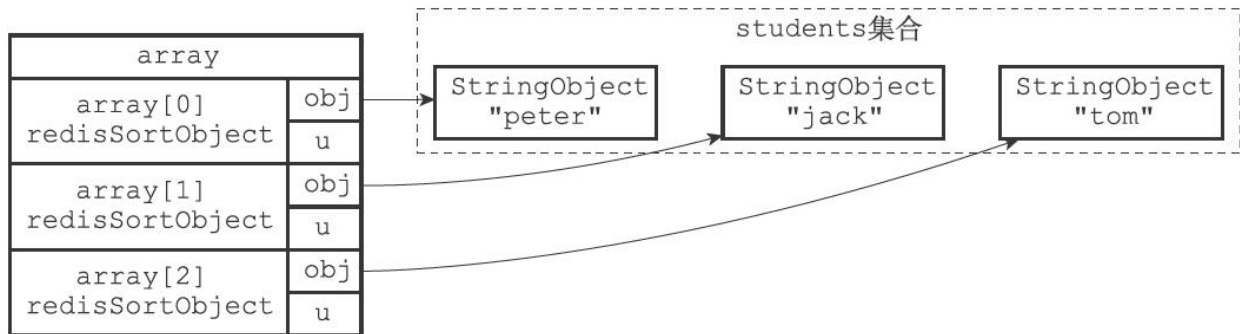


图21-17 数组结构

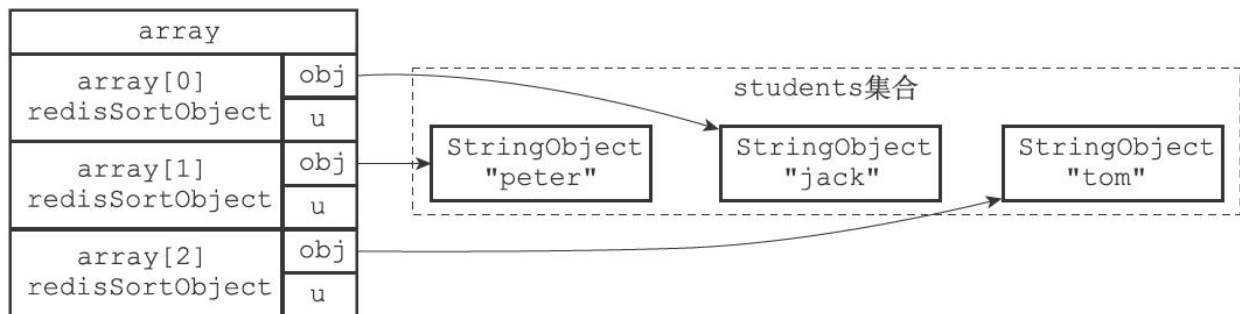


图21-18 数组结构

5个元素

·`key="jack-name" value="Jack Snow"`

·`key="peter-name" value="Peter White"`

·`key="tom-name" value="Tom Smith"`

redis> SORT students ALPHA GET \*-name GET \*-birth

redis>

redis> SORT students ALPHA GET \*-name GET \*-birth

redis>

---

#

redis>

redis> SET peter-birth 1995-6-7

OK

redis> SET tom-birth 1995-8-16

OK

redis> SET jack-birth 1995-5-24

OK

#

redis>

redis>

redis> SORT students ALPHA GET \*-name GET \*-birth

1) "Jack Snow"

2) "1995-5-24"

3) "Peter White"

4) "1995-6-7"

5) "Tom Smith"

6) "1995-8-16"

---

redis> SORT students ALPHA GET \*-name GET \*-birth

redis>

redis>

4) "1995-6-7" obj { "name": "Peter White", "birth": "1995-6-7" }

\*-birth

· "jack" { "name": "Jack Snow", "birth": "1995-5-24" }

·{"jack":{"\*-birth":"jack-birth"}}

·{"peter":{"\*-name":"peter-name"}}

·{"peter":{"\*-birth":"peter-birth"}}

·{"tom":{"\*-name":"tom-name"}}

·{"tom":{"\*-birth":"tom-birth"}}

5{"\*":"\*"}5

·{"jack-name":"Jack Snow"}

·{"jack-birth":"1995-5-24"}

·{"peter-name":"Peter White"}

·{"peter-birth":"1995-6-7"}

·{"tom-name":"Tom Smith"}

·{"tom-birth":"1995-8-16"}

**SORT**GET

## 21.8 STORE 命令

命令格式 SORT 命令

---

```
redis> SADD students "peter" "jack" "tom"
(integer) 3
redis> SORT students ALPHA
1) "jack"
2) "peter"
3) "tom"
```

---

命令格式 STORE 命令

命令

---

```
redis> SORT students ALPHA STORE sorted_students
(integer) 3
redis> LRANGE sorted_students 0-1
1) "jack"
2) "peter"
3) "tom"
```

---

命令格式 SORT students ALPHA STORE sorted\_students 命令

命令

1 redisSortObject 命令

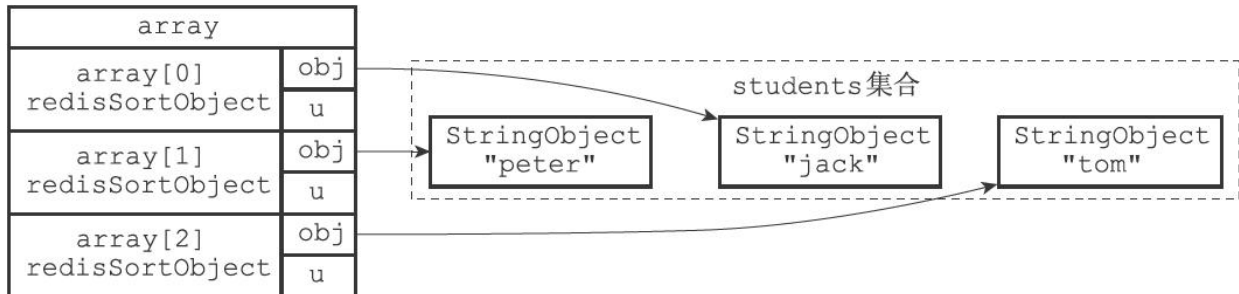
2 obj 命令

3 obj 命令

· 0 00000000 "jack" 0000

· 1 00000000 "peter" 0000

· 2 00000000 "tom" 0000



21-19 00000000

4 sorted\_students 000000000000000000000000

5 sorted\_students 00000000

6 0000000000000000 "jack" "peter" "tom" 0000

sorted\_students 0000000000000000 RPush

sorted\_students "jack" "peter" "tom"

7 0000000000000000 "jack" "peter" "tom" 0000

SORT 00000000 STORE 00000000000000000000000000000000

## 21.9 查詢數據

查詢數據的 `SORT` 子句用於對查詢結果進行排序。排序的默認方式是按升序排列。要按降序排列，則在 `SORT` 子句後面加上 `DESC`。要按多個列排序，則在 `SORT` 子句後面加上多個列名。

### 21.9.1 查詢數據

查詢數據的 `SORT` 子句用於對查詢結果進行排序。

1. 按列名排序。例如，按 `ALPHA` 列升序排列，按 `DESC` 降序排列，按 `BY` 列升序排列。

2. 按行數限制。例如，按 `LIMIT` 子句限制返回的行數。例如，`LIMIT 10` 表示返回前 10 行。

3. 按列名和行數限制。例如，按 `GET` 子句限制返回的行數。例如，`GET 10` 表示返回前 10 行。

4. 按列名和行數限制。例如，按 `STORE` 子句限制返回的行數。

5. 按列名和行數限制。例如，按 `STORE` 子句限制返回的行數。

□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□

---

SORT <key> ALPHA DESC BY <by-pattern> LIMIT <offset> <count> GET  
<get-pattern> STORE <store\_key>

---

□□□□□□□□□□

---

SORT <key> ALPHA DESC BY <by-pattern>

---

□□□□□

---

LIMIT <offset> <count>

---

□□□□□

---

GET <get-pattern>

---

□□□□□

---

STORE <store\_key>

---

□□□□□□□□□□□□□□□□□□□□□□□□



## 21.9.2 排序和过滤

排序和过滤操作是 SORT 和 GET 操作的一部分。

SORT 操作

语法

---

```
SORT <key> ALPHA DESC BY <by-pattern> LIMIT <offset> <count> GET
<get-pattern> STORE <store_key>
```

---

参数

---

```
SORT <key> LIMIT <offset> <count> BY <by-pattern> ALPHA GET <get-
pattern> STORE <store_key> DESC
```

---

参数

---

```
SORT <key> STORE <store_key> DESC BY <by-pattern> GET <get-pattern>
ALPHA LIMIT <offset> <count>
```

---

参数

排序和过滤操作是 SORT 和 GET 操作的一部分。

排序和过滤操作是 SORT 和 GET 操作的一部分。

参数

SORT <key> GET <pattern-a> GET <pattern-b> STORE <store\_key>

---

□□□□

---

SORT <key> STORE <store\_key> GET <pattern-a> GET <pattern-b>

---

□□□□□□□□□□□□□□□□GET□□□□□□□□□□

---

SORT <key> STORE <store\_key> GET <pattern-b> GET <pattern-a>

---

□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□SORT□□□□□□□□□□□□□□□□GET□□□

## 21.10 排序

·`SORT` 语句用于对查询结果进行排序。  
语法

·`ORDER BY` 语句用于对查询结果进行排序。

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。  
语法

·`ORDER BY` 语句支持 `NULLS FIRST` 和 `NULLS LAST` 两种排序方式。

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。  
语法  
排序

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。  
语法

·`ORDER BY` 语句支持 `ASC` 和 `DESC` 两种排序方式。

·`sort` `alpha asc`  
`desc by limit get store`

·`get sort`

## 22 位数组

Redis 的 SETBIT GETBIT BITCOUNT BITOP 命令

操作 bit array “”

SETBIT 命令将指定的 bit 设置为 0 或 1

返回 0 或 1

---

```
redis> SETBIT bit 0 1 # 0000 0001
(integer) 0
redis> SETBIT bit 3 1 # 0000 1001
(integer) 0
redis> SETBIT bit 0 0 # 0000 1000
(integer) 1
```

---

GETBIT 命令返回指定的 bit

---

```
redis> GETBIT bit 0 # 0000 1000
(integer) 0
redis> GETBIT bit 3 # 0000 1000
(integer) 1
```

---

BITCOUNT 命令返回指定的 bit 的个数

---

```
redis> BITCOUNT bit # 0000 1000
(integer) 1
redis> SETBIT bit 0 1 # 0000 1001
(integer) 0
redis> BITCOUNT bit
(integer) 2
```

---

```
redis> SETBIT bit 1 1 # 0000 1011
(integer) 0
redis> BITCOUNT bit
(integer) 3
```

---

BITOP and or xor

```
redis> SETBIT x 3 1 # x = 0000 1011
(integer) 0
redis> SETBIT x 1 1
(integer) 0
redis> SETBIT x 0 1
(integer) 0
redis> SETBIT y 2 1 # y = 0000 0110
(integer) 0
redis> SETBIT y 1 1
(integer) 0
redis> SETBIT z 2 1 # z = 0000 0101
(integer) 0
redis> SETBIT z 0 1
(integer) 0
redis> BITOP AND and-result x y z # 0000 0000
(integer) 1
redis> BITOP OR or-result x y z # 0000 1111
(integer) 1
redis> BITOP XOR xor-result x y z # 0000 1000
(integer) 1
```

---

not

```
redis> SETBIT value 0 1 # 0000 1001
(integer) 0
redis> SETBIT value 3 1
(integer) 0
redis> BITOP NOT not-value value # 1111 0110
(integer) 1
```

---

Redis GETBIT SETBIT  
BITCOUNT BITOP

## 22.1 SDS

Redis 使用 SDS 来存储字符串。SDS 是 Simple Dynamic String 的缩写。它是对 C 语言字符串的改进。SDS 是 Redis 字符串数据类型的基础。

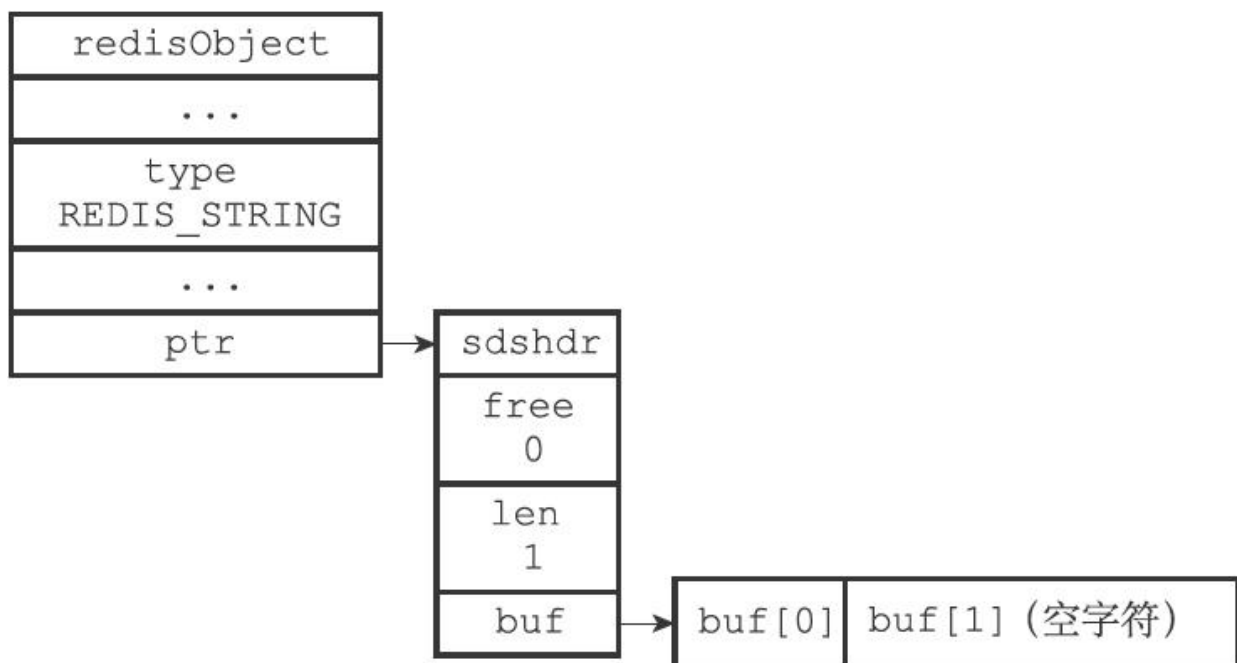
图 22-1 展示了 SDS 的数据结构。

· redisObject.type 指向 REDIS\_STRING

· sdshdr.len 指向 1，表示字符串的长度。

· buf 指向 buf[0]，指向字符串的起始位置。

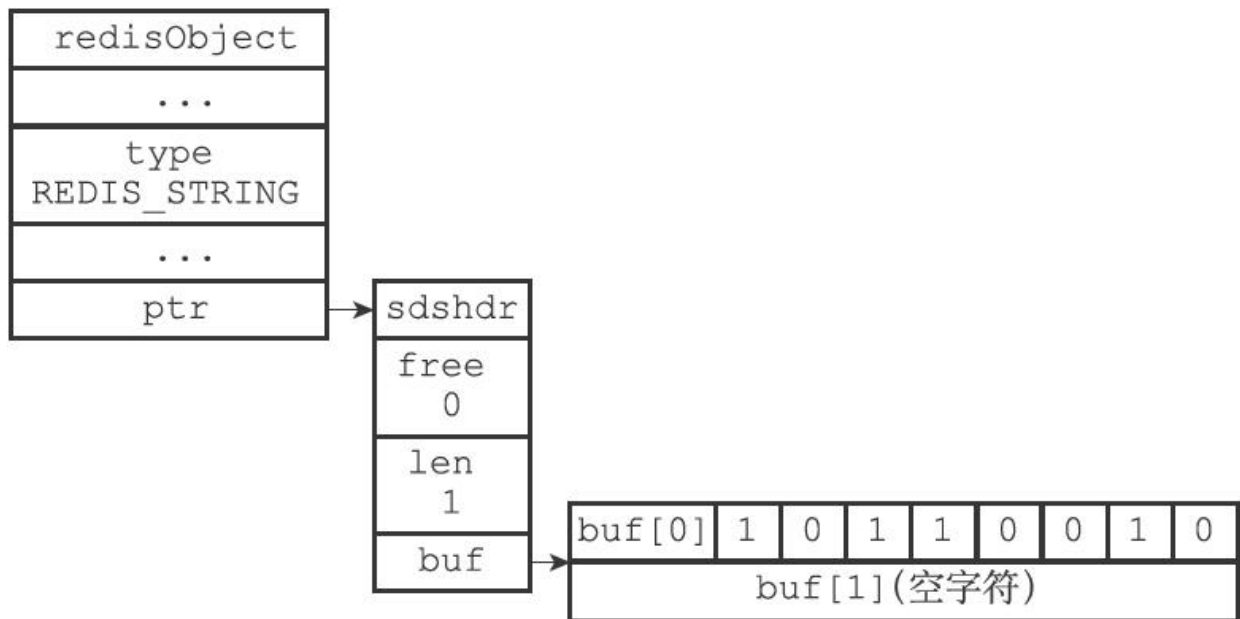
· buf 指向 buf[1]，指向字符串的结束位置（空字符 '\0'）。





## 图22-1 SDS的组成

图22-1展示了SDS的组成。在图22-1中，我们看到了一个指向SDS的指针，它指向了一个指向buf的指针。图22-2展示了SDS的组成。



## 图22-2 SDS的组成

图22-2展示了SDS的组成。在图22-2中，我们看到了一个指向buf的指针，它指向了一个指向buf[i]的指针。图22-2展示了SDS的组成。

图22-2展示了SDS的组成。在图22-2中，我们看到了一个指向buf的指针，它指向了一个指向buf[i]的指针。图22-2展示了SDS的组成。

图22-2展示了SDS的组成。在图22-2中，我们看到了一个指向buf的指针，它指向了一个指向buf[i]的指针。图22-2展示了SDS的组成。

图22-2展示了SDS的组成。在图22-2中，我们看到了一个指向buf的指针，它指向了一个指向buf[i]的指针。图22-2展示了SDS的组成。

图22-3 SDS数据结构

·sdshdr.len为3，表示SDS字符串的长度

·buf[0]buf[1]buf[2]为字符串的内存地址

1111 0000 1100 0011 1010 0101buf  
1010 0101 1100 0011 0000 1111

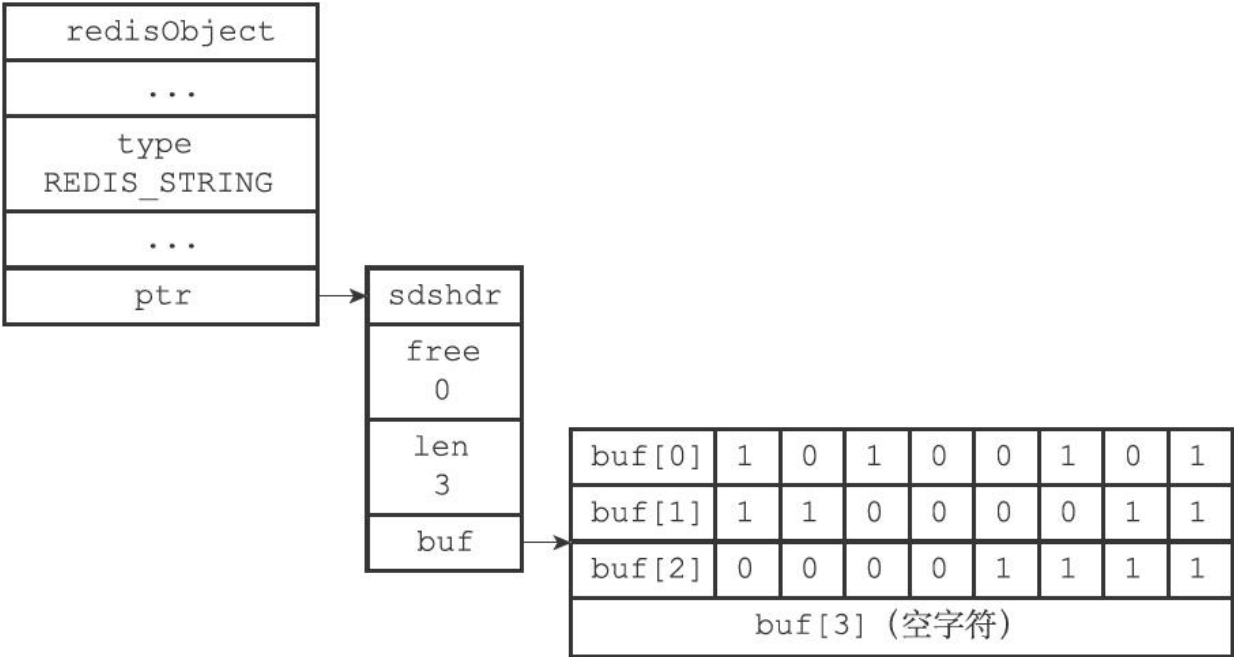


图22-3 SDS数据结构

## 22.2 GETBIT関数

GETBIT関数は、bitarrayのoffset番目のビットを返します。

---

GETBIT <bitarray> <offset>

---

GETBIT関数の引数は、

1番目のbyte =  $\lfloor \text{offset} \div 8 \rfloor$  byte番目のoffset番目のビットを返します。

2番目のbit =  $\text{offset} \bmod 8 + 1$  bit番目のoffset番目のビットを返します。

3番目のbyte番目のbit番目のbitarrayのoffset番目のビットを返します。

bitarrayの22-2番目のビットを返します。

---

GETBIT <bitarray> 3

---

bitarray

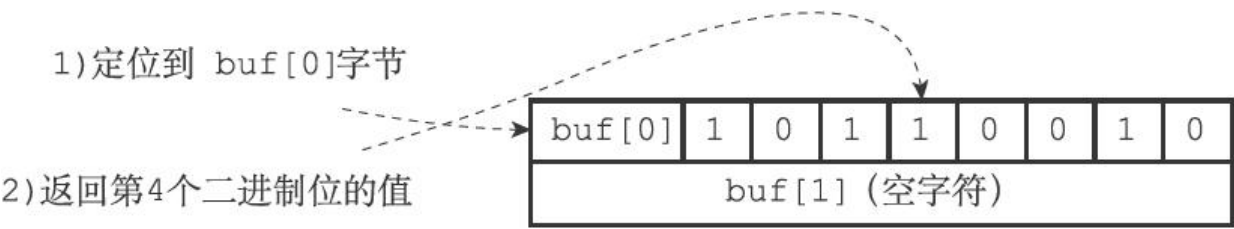
1  $\lfloor 3 \div 8 \rfloor = 0$

$$2 \div 3 \bmod 8 + 1 = 4$$

3) buf[0] 4

4) 1

22-4



22-4 offset 3

22-3

---

```
GETBIT <bitarray> 10
```

---

$$1 \div 10 \div 8 = 1$$

$$2 \div 10 \bmod 8 + 1 = 3$$

3) buf[1] 3

4) 0

图 22-5 图 22-5

1) 定位到buf[1] 字节

2) 返回第 3 个二进制位的值

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| buf[0] | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| buf[1] | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| buf[2] | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| buf[3] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

图 22-5 图 22-5 offset 10

GETBIT 0 1

## 22.3 SETBIT

SETBIT 関数は `bitarray` オブジェクトの `offset` 位置に `value` を設定し、`len` 以上の位置は 0 に設定します。

---

SETBIT <bitarray> <offset> <value>

---

1. SETBIT の引数

1. `len` =  $\lfloor \text{offset} \div 8 \rfloor + 1$  `len` 以上の位置は 0 に設定します。

2. `bitarray` オブジェクトは SDS オブジェクトの `len` 以上の位置は SDS オブジェクトの `len` 以上の位置は 0 に設定します。

3. `byte` =  $\lfloor \text{offset} \div 8 \rfloor$  `byte` 以上の位置は 0 に設定します。

4. `bit` =  $\text{offset} \bmod 8 + 1$  `bit` 以上の位置は `byte` 以上の位置は 0 に設定します。

5. `byte` `bit` `bitarray` オブジェクトの `offset` 位置に `oldvalue` を `value` に設定します。

6. `oldvalue`

`SETBIT` 0 1

### 22.3.1 SETBIT

`SETBIT`

22-2

---

`SETBIT <bitarray> 1 1`

---

1.  $1 \div 8 + 1$

2. SDS 1

3.  $1 \div 8$  0 1 `buf[0]`

4.  $1 \bmod 8 + 1$  2 1 `buf[0]` 2

5. `buf[0]` 2 0 `oldvalue`

1

6. `oldvalue` 0

- 1) 定位到buf[0]字节
- 2) 定位到buf[0]字节的第2个二进制位  
将位现在的值0保存到oldvalue变量  
然后将位的值设置为1

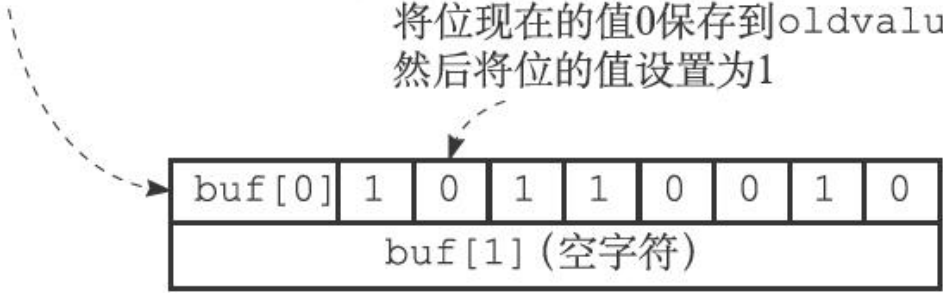


图22-6 SETBIT操作前

图22-6 对SETBIT操作前图22-7 对SETBIT操作后

图



图22-7 SETBIT操作后

### 22.3.2 对SETBIT操作

对SETBIT操作

图22-2

```
SETBIT <bitarray> 12 1
```

图



1.  $12 \div 8 + 1 = 2$ ，所以申请 2 个字节的空间。

2. 申请 2 个字节的空间，1 个字节用于存储字符串，2 个字节用于存储 SDS 结构体。SDS 结构体包含 2 个字节的空间，1 个字节用于存储 buf 指针，5 个字节用于存储 22-8 的值。

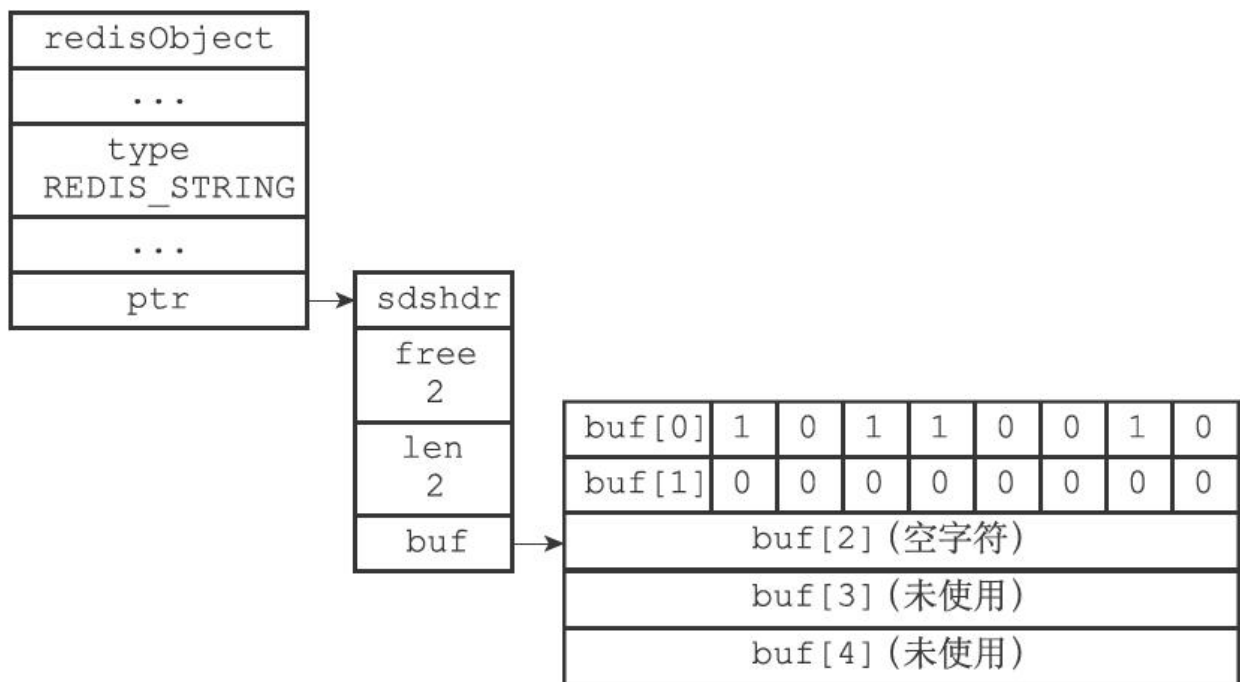


图 22-8 内存布局

3.  $12 \div 8 = 1$ ，所以 buf[1] 的值是 1。

4.  $12 \bmod 8 + 1 = 5$ ，所以 buf[1] 的值是 5。



## 例22-10 SETBIT関数の動作

関数bufは、メモリ上に連続したバイト配列を返す。bufは、メモリ上に連続したバイト配列を返す。

関数bufは、メモリ上に連続したバイト配列を返す。bufは、メモリ上に連続したバイト配列を返す。SETBIT関数は、bufの指定したビットを1に設定する。CPUは、bufの指定したビットを1に設定する。

例22-11例22-14は、bufの指定したビットを1に設定する。0100 1101は、SETBIT <bitarray> 12 1の指定したビットを1に設定する。0001 0000 0100 1101は、SETBIT <bitarray> 12 1の指定したビットを1に設定する。

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| buf[1] (空文字) |   |   |   |   |   |   |   |   |

## 例22-11 SETBIT関数の動作

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| buf[1]       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| buf[2] (空文字) |   |   |   |   |   |   |   |   |
| buf[3] (未使用) |   |   |   |   |   |   |   |   |
| buf[4] (未使用) |   |   |   |   |   |   |   |   |

## 例22-12 SETBIT関数の動作

将字节buf[0]的所有二进制位  
移动到字节buf[1]

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| buf[1]       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| buf[2] (空字符) |   |   |   |   |   |   |   |   |
| buf[3] (未使用) |   |   |   |   |   |   |   |   |
| buf[4] (未使用) |   |   |   |   |   |   |   |   |

图22-13 移位操作

将偏移量为12的二进制位  
的值设置为1

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| buf[1]       | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| buf[2] (空字符) |   |   |   |   |   |   |   |   |
| buf[3] (未使用) |   |   |   |   |   |   |   |   |
| buf[4] (未使用) |   |   |   |   |   |   |   |   |

图22-14 设置特定位

## 22.4 BITCOUNT

BITCOUNT 1

22-15 BITCOUNT 4

22-16 BITCOUNT 12

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| buf[1] (空字符) |   |   |   |   |   |   |   |   |

22-15 BITCOUNT

|              |   |   |   |   |   |   |   |   |
|--------------|---|---|---|---|---|---|---|---|
| buf[0]       | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| buf[1]       | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| buf[2]       | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| buf[3] (空字符) |   |   |   |   |   |   |   |   |

22-16 BITCOUNT

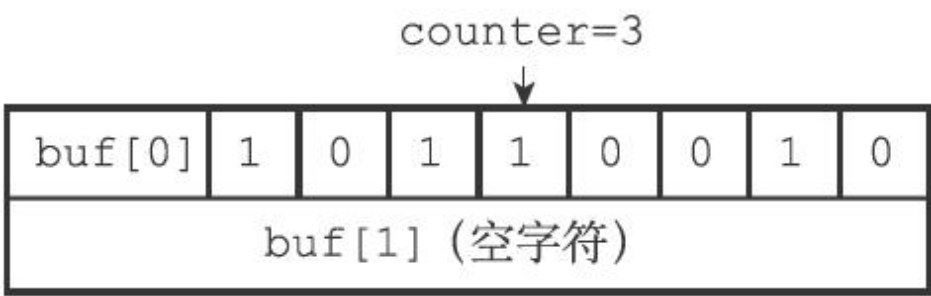
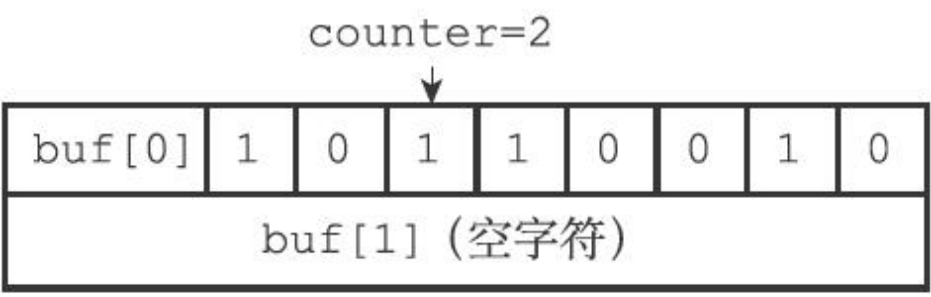
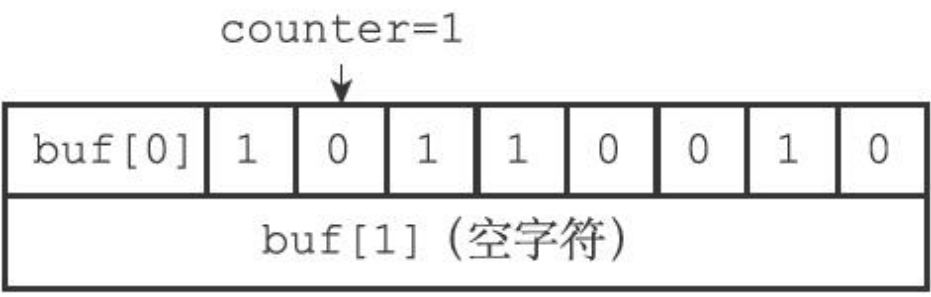
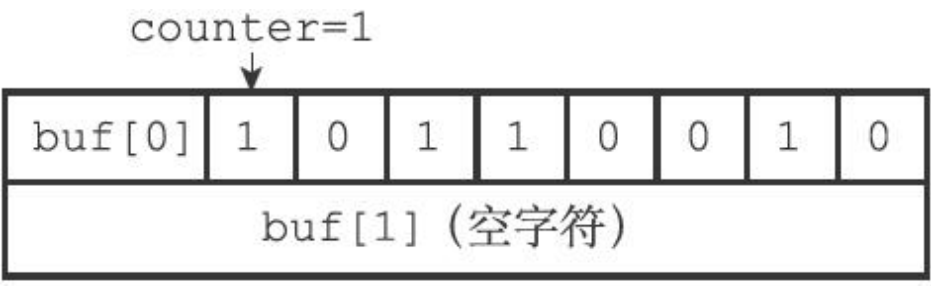
BITCOUNT

BITCOUNT BITCOUNT

22.4.1 统计字符串中1的个数

使用BITCOUNT统计字符串中1的个数

22-17 统计字符串中1的个数









| 键（位数组）    | 值（值为 1 的位数量） |
|-----------|--------------|
| 0000 0000 | 0            |
| 0000 0001 | 1            |
| 0000 0010 | 1            |
| 0000 0011 | 2            |
| 0000 0100 | 1            |
| 0000 0101 | 2            |
| 0000 0110 | 2            |
| 0000 0111 | 3            |
| ...       | ...          |
| 1111 1101 | 7            |
| 1111 1110 | 7            |
| 1111 1111 | 8            |

键的个数为  $2^2-1$ ，即 3，因此共有 8 个键。每个键的值为 0 到 7 之间的整数。

100MB = 800000000bit。因此，100MB 的键的个数为 800000000。

· 500MB

· 16 16 100MB

500MB

· 32 32 100MB

500MB

·

8

16 KB 32 GB

KB

· CPU CPU

CPU cache

miss

8 16

BITCOUNT 是 C 语言中为数不多的特殊函数之一，它返回一个整数的二进制表示中 1 的个数。在 C 语言中，BITCOUNT 函数通常用于计算一个整数的 Hamming Weight。在 C 语言中，variable-precision SWAR 是一种高效的并行位操作技术，它可以在一个 32 位的寄存器中同时操作多个 1 位的子问题。

### 22.4.3 32 位的 variable-precision SWAR

BITCOUNT 函数——返回一个整数的二进制表示中 1 的个数。在 C 语言中，Hamming Weight 是指一个整数的二进制表示中 1 的个数。

在 C 语言中，variable-precision SWAR 是一种高效的并行位操作技术，它可以在一个 32 位的寄存器中同时操作多个 1 位的子问题。在 C 语言中，variable-precision SWAR 是一种高效的并行位操作技术，它可以在一个 32 位的寄存器中同时操作多个 1 位的子问题。

32 位的 variable-precision SWAR

---

```
uint32_t swar(uint32_t i) {
 //
 1
 i = (i & 0x55555555) + ((i >> 1) & 0x55555555);
 //
 2
 i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
 //
 3
 i = (i & 0x0F0F0F0F) + ((i >> 4) & 0x0F0F0F0F);
 //
 4
 i = (i * (0x01010101) >> 24);
 return i;
}
```

---

对swar对bitarray

·对1对i

·对2对i

·对3对i

·对4对i\*0x01010101对bitarray

>>24对bitarray对bitarray

对swar对0x3A70F21B

0x2560A116对0x3A70F21B

对22-2

22-2 对0x3A70F21B

| 值          | 分 组 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|------------|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0x3A70F21B | 00  | 11 | 10 | 10 | 01 | 11 | 00 | 00 | 11 | 11 | 00 | 10 | 00 | 01 | 10 | 11 |
| 0x2560A116 | 00  | 10 | 01 | 01 | 01 | 10 | 00 | 00 | 10 | 10 | 00 | 01 | 00 | 01 | 01 | 10 |
| 汉明重量       | 0   | 2  | 1  | 1  | 1  | 2  | 0  | 0  | 2  | 2  | 0  | 1  | 0  | 1  | 1  | 2  |

0x223041130x3A70F21B22-3

22-3 0x3A70F21B

| 值          | 分 组  |      |      |      |      |      |      |      |
|------------|------|------|------|------|------|------|------|------|
| 0x3A70F21B | 0011 | 1010 | 0111 | 0000 | 1111 | 0010 | 0001 | 1011 |
| 0x22304113 | 0010 | 0010 | 0011 | 0000 | 0100 | 0001 | 0001 | 0011 |
| 汉明重量       | 2    | 2    | 3    | 0    | 4    | 1    | 1    | 3    |

0x40305040x3A70F21B22-4

22-4 0x3A70F21B

| 值          | 分 组      |          |          |          |
|------------|----------|----------|----------|----------|
| 0x3A70F21B | 00111010 | 01110000 | 11110010 | 00011011 |
| 0x4030504  | 00000100 | 00000011 | 00000101 | 00000100 |
| 汉明重量       | 4        | 3        | 5        | 4        |

0x4030504\*0x01010101=0x100c090422-5

22-5 0x3A70F21B0x100c0904

| 值          | 24 位至 31 位 | 16 至 23 位 | 8 至 15 位 | 0 至 7 位  |
|------------|------------|-----------|----------|----------|
| 0x100c0904 | 00010000   | 00001100  | 00001001 | 00000100 |
| 汉明重量       | 16         | 无用值       | 无用值      | 无用值      |

0x100c0904 >> 24 0x10  
16 0x3A70F21B 22-6

22-6 0x3A70F21B

| 值    | 24 位至 31 位 | 16 至 23 位 | 8 至 15 位 | 0 至 7 位  |
|------|------------|-----------|----------|----------|
| 0x10 | 00000000   | 00000000  | 00000000 | 00010000 |

swar 32 32  
8 4 16 2 swar

swar  
swar

· swar  
32 64

· swar 128  
swar

swar

## 22.4.4 4 Redis

BITCOUNT variable-precisionSWAR

· 8 0000 0000 1111 1111

· variable-precision SWAR BITCOUNT  
128 32 variable-precision SWAR 128

BITCOUNT

· 128 variable-precision  
SWAR

· 128

BITCOUNT

---

```


8

0000 0011
3

weight_in_byte[3]
2
2
0000 0011

weight_in_byte = [0,1,1,2,1,2,2,/*...*/7,7,8]
```

```

def BITCOUNT(bits):
 #
 #####
 count = count_bit(bits)
 #
 #####
 weight = 0
 #
 #####128
 []
 #
 #####variable-precision SWAR
 #####
 while count >= 128:
 #
 []swar
 #####32
 #####
 #
 []bits[i:j]
 []j
 #####
 weight += swar(bits[0:32])
 weight += swar(bits[32:64])
 weight += swar(bits[64:96])
 weight += swar(bits[96:128])
 #
 #####
 bits = bits[128:]
 #
 #####
 count -= 128
 #
 #####128
 []
 #
 #####
 while count:
 #
 []8
 #####
 index = bits_to_unsigned_int(bits[0:8])
 weight += weight_in_byte[index]
 #
 #####
 bits = bits[8:]
 #
 #####

```



```

 count -= 8
 #
 return weight

```

---

BITCOUNT 0 n

BITCOUNT n

loop<sub>1</sub> = n ÷ 128

loop<sub>2</sub> = n mod 128

100MB = 800000000bit BITCOUNT 100MB

500MB = 4000000000bit

BITCOUNT 500MB

100MB 500MB

## 22.5 BITOP□□□□□

00C0000000000000&000000|000000^000000~000000  
 0BITOP000AND0OR0XOR0NOT00000000000000000000

· BITOP AND &

· BITOP OR

```
·BITOP XOR^

```

・BITOP NOTは、ビット反転を意味する。

□ □ □ □ □ □ □ □ □ □ □ □ □ □

BITOP AND result x y

0000x0000000022-1800000y0000000022-19000BITOP00  
 00000000

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| buf[0] | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| buf[1] | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| buf[2] | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| buf[3] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

图22-18 输入x的初始值

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| buf[0] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| buf[1] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| buf[2] | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| buf[3] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

图22-19 输入y的初始值

1 将value[0]与buf[0]进行AND操作

2 将buf[0]与buf[0]进行AND操作，得到value[0]

□□

3 将value[1]与buf[1]进行AND操作，得到value[1]

□□

4 将value[2]与buf[2]进行AND操作，得到value[2]

□□

5 22-20 result

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| buf[0] | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| buf[1] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| buf[2] | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| buf[3] | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

22-20 x y BITOP AND

BITOP OR BITOP XOR BITOP NOT BITOP  
AND

BITOP AND BITOP OR BITOP XOR  
O n2  
BITOP NOT O n

22.6 □□□□

- Redis SDS

```
·SDS[0][0][0][0][0][0][0][0][0][0][0][0]SETBIT[0][0][0][0][0]SETBIT[0][0][0]
[0][0][0][0][0][0][0][0][0][0][0][0][0][0][0][0]
```

·BITCOUNT variable-precision SWAR

```
·BITOP□□□□□□□□□□C□□□□□□□□□□□□
```

## 22.7 面试题

·StackOverflow面试题Hamming Weight面试题  
面试题<http://stackoverflow.com/questions/109023/how-to-count-the-number-of-set-bits-in-a-32-bit-integer>

·面试题Counting The Number Of Set Bits In An Integer  
面试题variable-precision SWAR面试题  
<http://yesteapea.wordpress.com/2013/03/03/counting-the-number-of-set-bits-in-an-integer/>

## 23 配置

Redis 默认情况下，慢查询日志是关闭的。要开启慢查询日志，需要配置以下参数：

slowlog-log-slower-than

slowlog-log-slower-than 1000000 1000000  
000 表示慢查询日志的阈值，单位是微秒。默认值是 1000000，即 1 秒。

slowlog-max-len 100 表示慢查询日志的最大长度。默认值是 100，即最多保存 100 条慢查询记录。

slowlog-max-len

slowlog-max-len 100 表示慢查询日志的最大长度。默认值是 100，即最多保存 100 条慢查询记录。

slowlog-max-len 100 表示慢查询日志的最大长度。默认值是 100，即最多保存 100 条慢查询记录。

CONFIG SET slowlog-log-slower-than 0 表示关闭慢查询日志。Redis 默认情况下，慢查询日志是关闭的。

slowlog-max-len 5

---

```
redis> CONFIG SET slowlog-log-slower-than 0
OK
redis> CONFIG SET slowlog-max-len 5
OK
```

---

---

```
redis> SET msg "hello world"
OK
redis> SET number 10086
OK
redis> SET database "Redis"
OK
```

---

SLOWLOG GET

---

```
redis> SLOWLOG GET
1) 1)
integer
4 #
uid
2) (integer) 1378781447 #
UNIX
3) (integer) 13 #
4) 1) "SET" #
"database"
"Redis"
2) 1) (integer) 3
2) (integer) 1378781439
3) (integer) 10
4) 1) "SET"
2) "number"
```



```
3) "10086"
3) 1) (integer) 2
 2) (integer) 1378781436
 3) (integer) 18
4) 1) "SET"
 2) "msg"
 3) "hello world"
4) 1) (integer) 1
 2) (integer) 1378781425
 3) (integer) 11
4) 1) "CONFIG"
 2) "SET"
 3) "slowlog-max-len"
 4) "5"
5) 1) (integer) 0
 2) (integer) 1378781415
 3) (integer) 53
4) 1) "CONFIG"
 2) "SET"
 3) "slowlog-log-slower-than"
 4) "0"
```

---

```
00000000SLOWLOG GET00000000000000000000SLOWLOG
GET000000000000000000000000ID00000000000000000000
0000500
```

---

```
redis> SLOWLOG GET
1) 1) (integer) 5
 2) (integer) 1378781521
 3) (integer) 61
 4) 1) "SLOWLOG"
 2) "GET"
2) 1) (integer) 4
 2) (integer) 1378781447
 3) (integer) 13
 4) 1) "SET"
 2) "database"
 3) "Redis"
3) 1) (integer) 3
 2) (integer) 1378781439
 3) (integer) 10
 4) 1) "SET"
```

- 2) "number"
  - 3) "10086"
  - 4) 1) (integer) 2
    - 2) (integer) 1378781436
    - 3) (integer) 18
    - 4) 1) "SET"
      - 2) "msg"
      - 3) "hello world"
  - 5) 1) (integer) 1
    - 2) (integer) 1378781425
    - 3) (integer) 11
    - 4) 1) "CONFIG"
      - 2) "SET"
      - 3) "slowlog-max-len"
      - 4) "5"
-

## 23.1 慢查询日志

慢查询日志是 Redis 中用于记录执行时间超过指定阈值的命令的日志。

---

```
struct redisServer {
 // ...
 //
 慢查询日志 ID
 long long slowlog_entry_id;
 //
 慢查询日志列表
 list *slowlog;
 //
 慢查询日志记录时间比指定时间慢多少
 long long slowlog_log_slower_than;
 //
 慢查询日志记录最大长度
 unsigned long slowlog_max_len;
 // ...
};
```

---

slowlog\_entry\_id 是慢查询日志记录的 ID，从 0 开始递增。id 是慢查询日志记录的 ID。

慢查询日志记录的时间比指定时间慢多少，slowlog\_log\_slower\_than 是慢查询日志记录的时间比指定时间慢多少。slowlog\_max\_len 是慢查询日志记录的最大长度。ID 是慢查询日志记录的 ID。

slowlog

slowlogEntry

---

```
typedef struct slowlogEntry {
 //
 long long id;
 //
 UNIX
 time_t time;
 //
 long long duration;
 //
 robj **argv;
 //
 int argc;
} slowlogEntry;
```

---

- 
- 1) (integer) 3
  - 2) (integer) 1378781439
  - 3) (integer) 10
  - 4) 1) "SET"
    - 2) "number"
    - 3) "10086"
- 

23-1

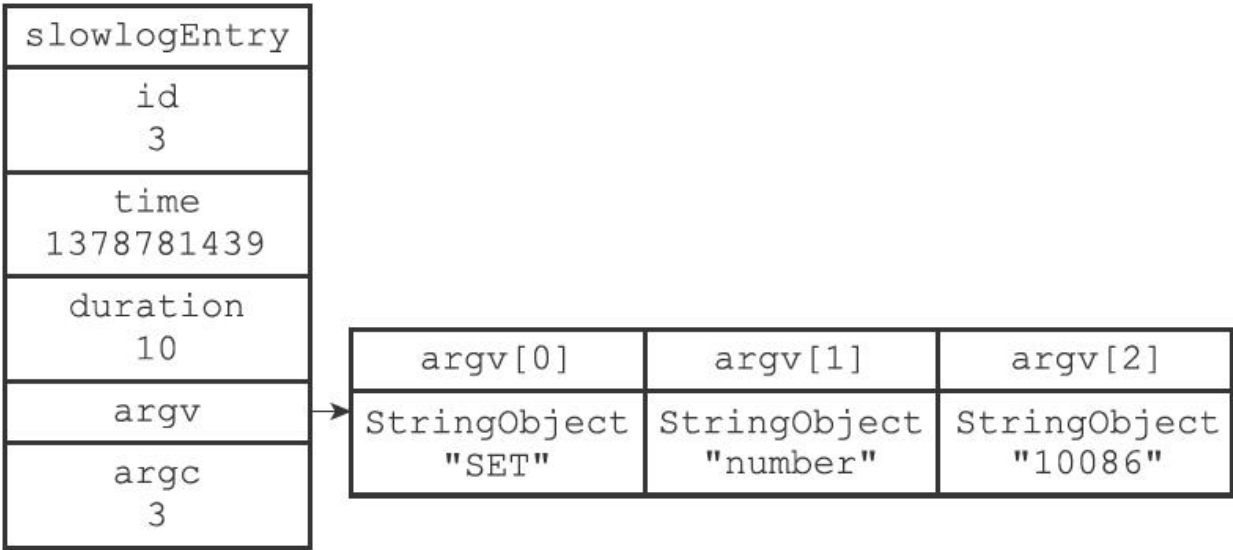


图23-1 slowlogEntry对象

图23-2 慢查询日志的存储结构

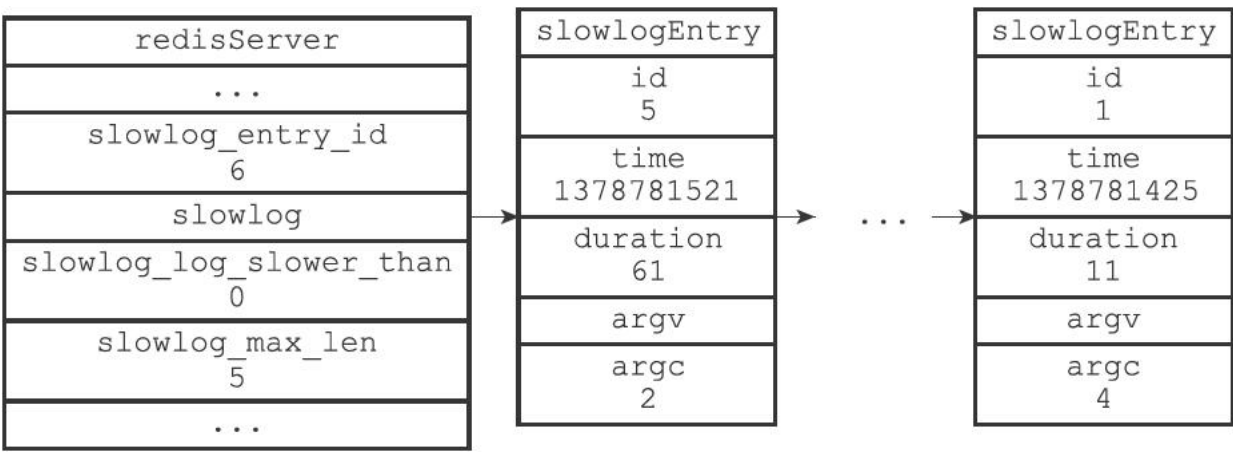


图23-2 redisServer对象

·slowlog\_entry\_id为6，表示慢查询日志的id为6

·slowlog为指向id为5的慢查询日志的指针，表示慢查询日志的id为5

慢查询日志的存储结构



## 23.2 慢查询日志

慢查询日志slowlog是Redis中记录慢查询的日志。

### SLOWLOG GET

---

```
def SLOWLOG_GET(number=None):
 #
 # number
 #
 #
 if number is None:
 number = SLOWLOG_LEN()
 #
 for log in redisServer.slowlog:
 if number <= 0:
 #
 break
 else:
 #
 number -= 1
 #
 printLog(log)
```

---

慢查询日志SLOWLOG LEN

---

```
def SLOWLOG_LEN():
 # slowlog
 return len(redisServer.slowlog)
```

---

## Redis SLOWLOG RESET 命令

---

```
def SLOWLOG_RESET():
 #
 # 清除 Redis 慢日志
 for log in redisServer.slowlog:
 #
 # 删除慢日志
 deleteLog(log)
```

---



## 23.3 慢查询

慢查询日志是 Redis 2.6 版本引入的一个新特性，它可以将那些执行时间超过指定阈值的命令记录到慢查询日志中。慢查询日志的格式如下：

```
slowlogPushEntryIfNeeded [threshold] slowlogPushEntryIfNeeded [threshold]
```

慢查询日志的格式如下：

---

```
#
慢查询日志
before = unixtime_now_in_us()
#
命令
execute_command(argv, argc, client)
#
慢查询日志
after = unixtime_now_in_us()
#
慢查询日志
slowlogPushEntryIfNeeded(argv, argc, before-after)
```

---

`slowlogPushEntryIfNeeded` 函数

1. 慢查询日志的阈值 `slowlog-log-slower-than` 配置项，单位为微秒。

慢查询日志的阈值 `slowlog-log-slower-than` 配置项，单位为微秒。

2. 慢查询日志的最大长度 `slowlog-max-len` 配置项，单位为条数。

慢查询日志的最大长度 `slowlog-max-len` 配置项，单位为条数。

慢查询日志的函数 `slowlogPushEntryIfNeeded`

---

```

void slowlogPushEntryIfNeeded(robj **argv, int argc, long long duration) {
 //
 // 如果慢日志已经满了，就不需要再推入了
 if (server.slowlog_log_slower_than < 0) return;
 //
 // 如果慢日志已经满了，就不需要再推入了
 if (duration >= server.slowlog_log_slower_than)
 //
 // 如果慢日志已经满了，就不需要再推入了
 listAddNodeHead(server.slowlog,slowlogCreateEntry(argv,argc,duration));
 //
 // 如果慢日志已经满了，就不需要再推入了
 while (listLength(server.slowlog) > server.slowlog_max_len)
 listDelNode(server.slowlog,listLast(server.slowlog));
}

```

---

慢日志的创建函数slowlogCreateEntry，它接收一个redisServer.slowlog\_entry\_id的指针，并返回一个慢日志的指针。

慢日志的创建函数slowlogCreateEntry，它接收一个redisServer.slowlog\_entry\_id的指针，并返回一个慢日志的指针。

---

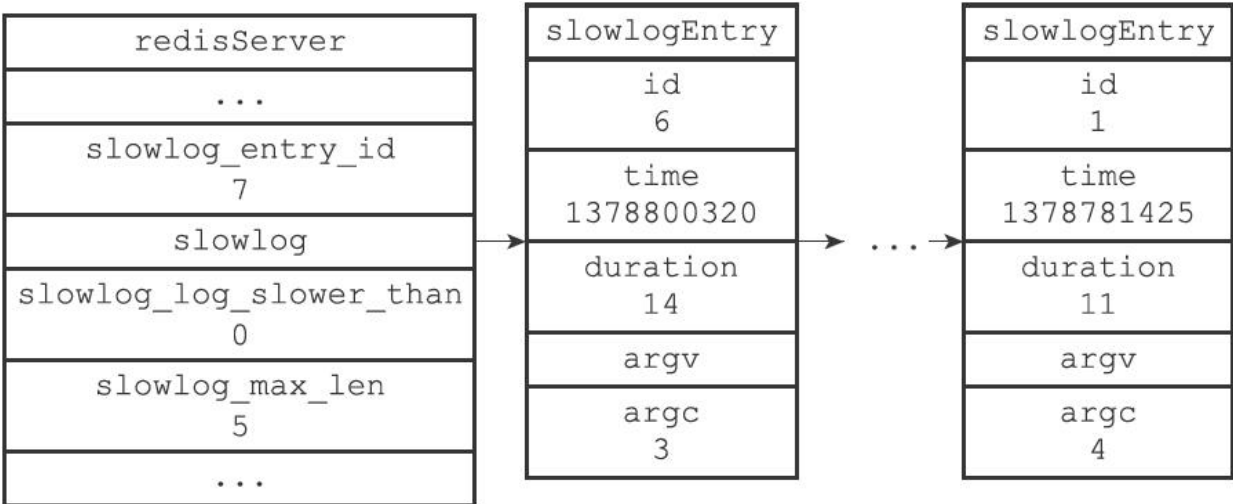
```

redis> EXPIRE msg 10086
(integer) 1

```

---

慢日志的创建函数slowlogCreateEntry，它接收一个redisServer.slowlog\_entry\_id的指针，并返回一个慢日志的指针。



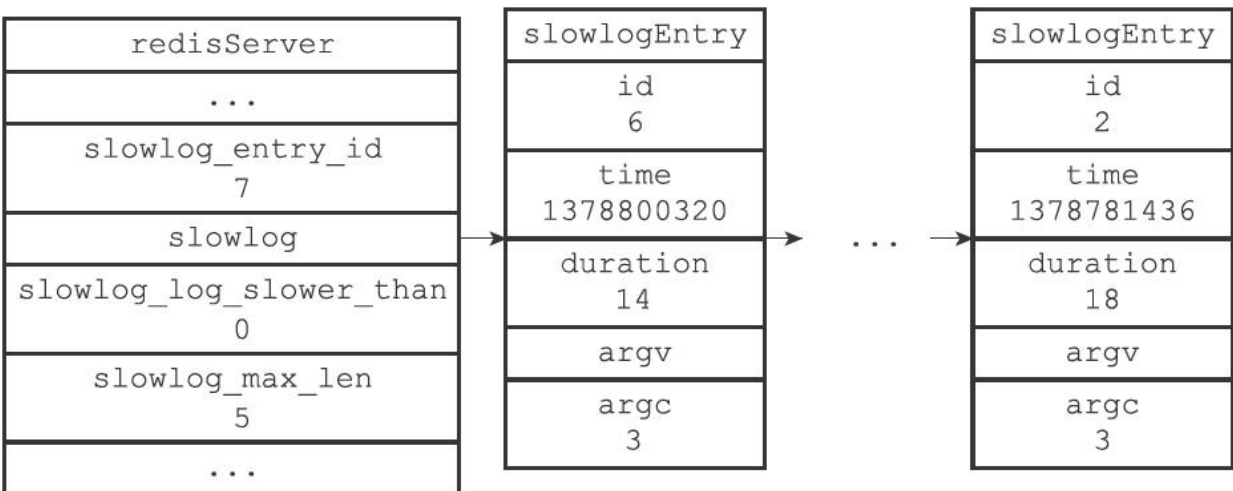
### 图23-3 EXPIRE的调用过程

redisServer的slowlog数组的slowlog\_entry\_id从6变为7

调用slowlogPushEntryIfNeeded函数，将慢查询记录推入慢查询数组

慢查询数组的slowlog\_entry\_id从6变为7，慢查询数组的slowlog\_log\_slower\_than从0变为5

### 图23-4 EXPIRE的调用过程



23-4 id1

## 23.4 Redis

- Redis

- Redis slowlog

slowlogEntry

- slowlog

- slowlog

- slowlog-max-len

## 24 实验

实验24-MONITOR实验环境搭建与实验操作

实验目的

---

```
redis> MONITOR
OK
1378822099.421623 [0 127.0.0.1:56604] "PING"
1378822105.089572 [0 127.0.0.1:56604] "SET" "msg" "hello world"
1378822109.036925 [0 127.0.0.1:56604] "SET" "number" "123"
1378822140.649496 [0 127.0.0.1:56604] "SADD" "fruits" "Apple" "Banana"
"Cherry"
1378822154.117160 [0 127.0.0.1:56604] "EXPIRE" "msg" "10086"
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
1378822258.690131 [0 127.0.0.1:56604] "DBSIZE"
```

---

实验步骤

实验环境搭建与实验操作24-1

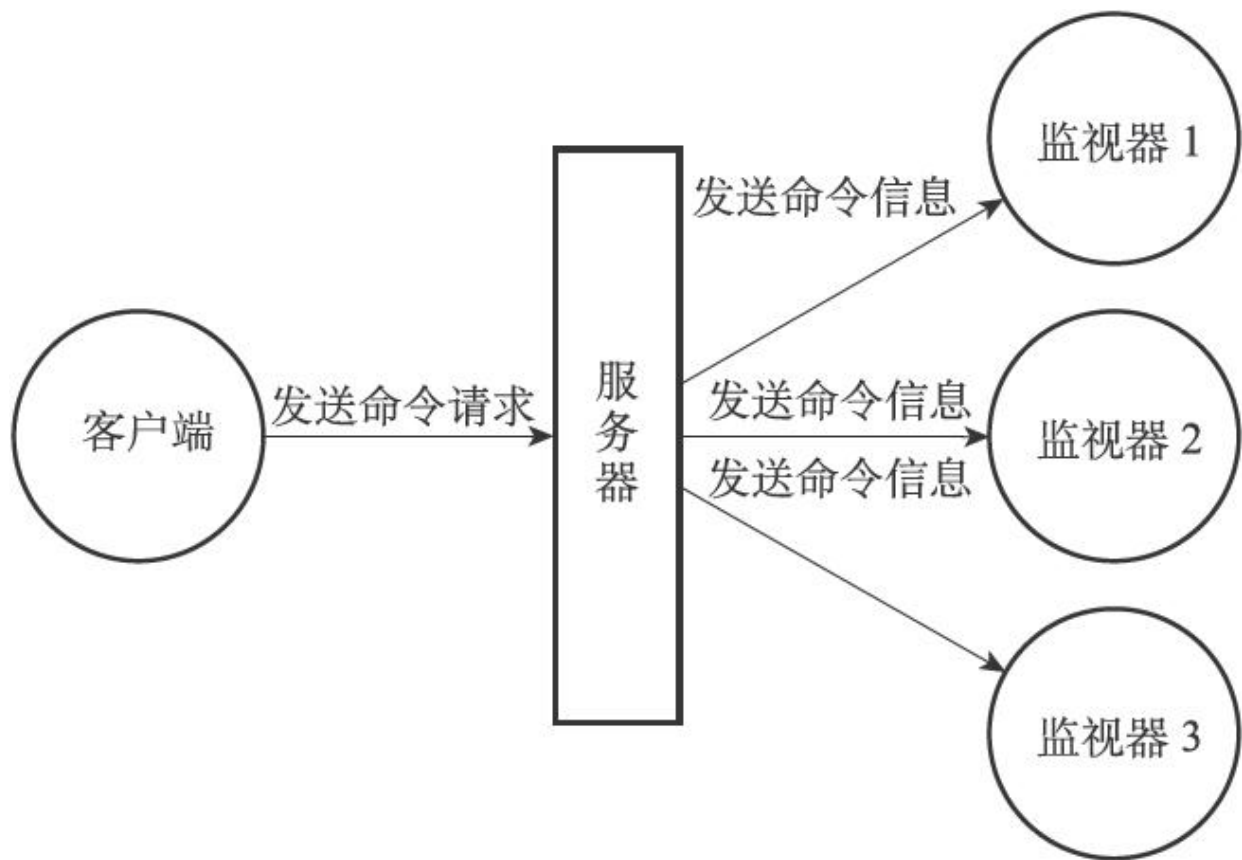


图24-1 系统结构示意图

## 24.1 哨兵

哨兵MONITOR命令用于监控Redis主从复制状态。

哨兵

---

```
def MONITOR():
 #
 # 哨兵命令
 client.flags |= REDIS_MONITOR
 #
 # 哨兵命令
 # 哨兵
 server.monitors.append(client)
 #
 # 哨兵命令OK
 send_reply("OK")
```

---

哨兵c10086命令MONITOR命令用于监控Redis主从复制状态。

REDIS\_MONITOR命令用于监控Redis主从复制状态。

哨兵c10086命令MONITOR命令用于监控Redis主从复制状态。

哨兵c10086命令MONITOR命令用于监控Redis主从复制状态。

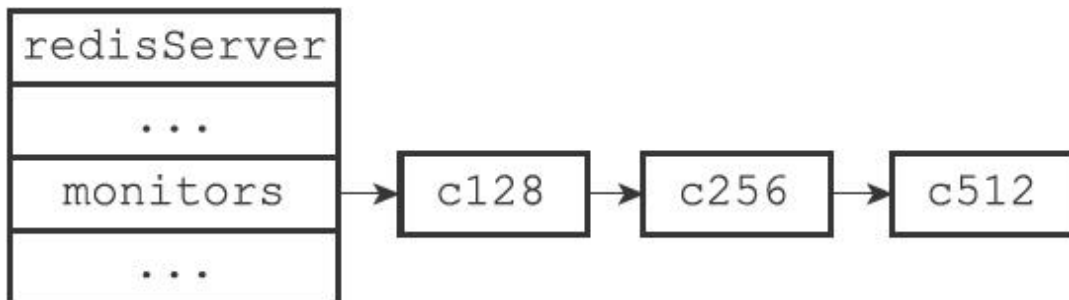




图24-2 在c10086中MONITOR命令的监视器

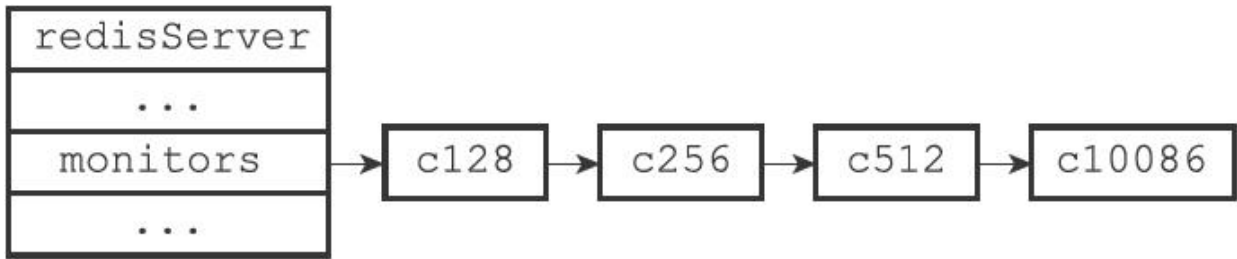


图24-3 在c10086中MONITOR命令的监视器

## 24.2 复制数据

复制数据时，需要向每个从节点发送 replicationFeedMonitors 消息，以通知其开始复制。

在 replicationFeedMonitors 消息中，需要包含以下信息：

---

```
def replicationFeedMonitors(client, monitors, dbid, argv, argc):
 #
 # 创建消息
 #
 msg = create_message(client, dbid, argv, argc)
 #
 for monitor in monitors:
 #
 send_message(monitor, msg)
```

---

例如，假设主节点的 IP 地址为 1378822257.329412，从节点的 IP 地址为 127.0.0.1，从节点的端口号为 56604，那么复制消息的格式如下：

---

```
1378822257.329412 [0 127.0.0.1:56604] "KEYS" "*"
```

---

消息中的 monitors 列表包含所有从节点的 ID。例如，如果从节点的 ID 为 24-3，那么消息中的 c128 字段的值为 c256，c512 字段的值为 c10086，c24-4 字段的值为 24-4。

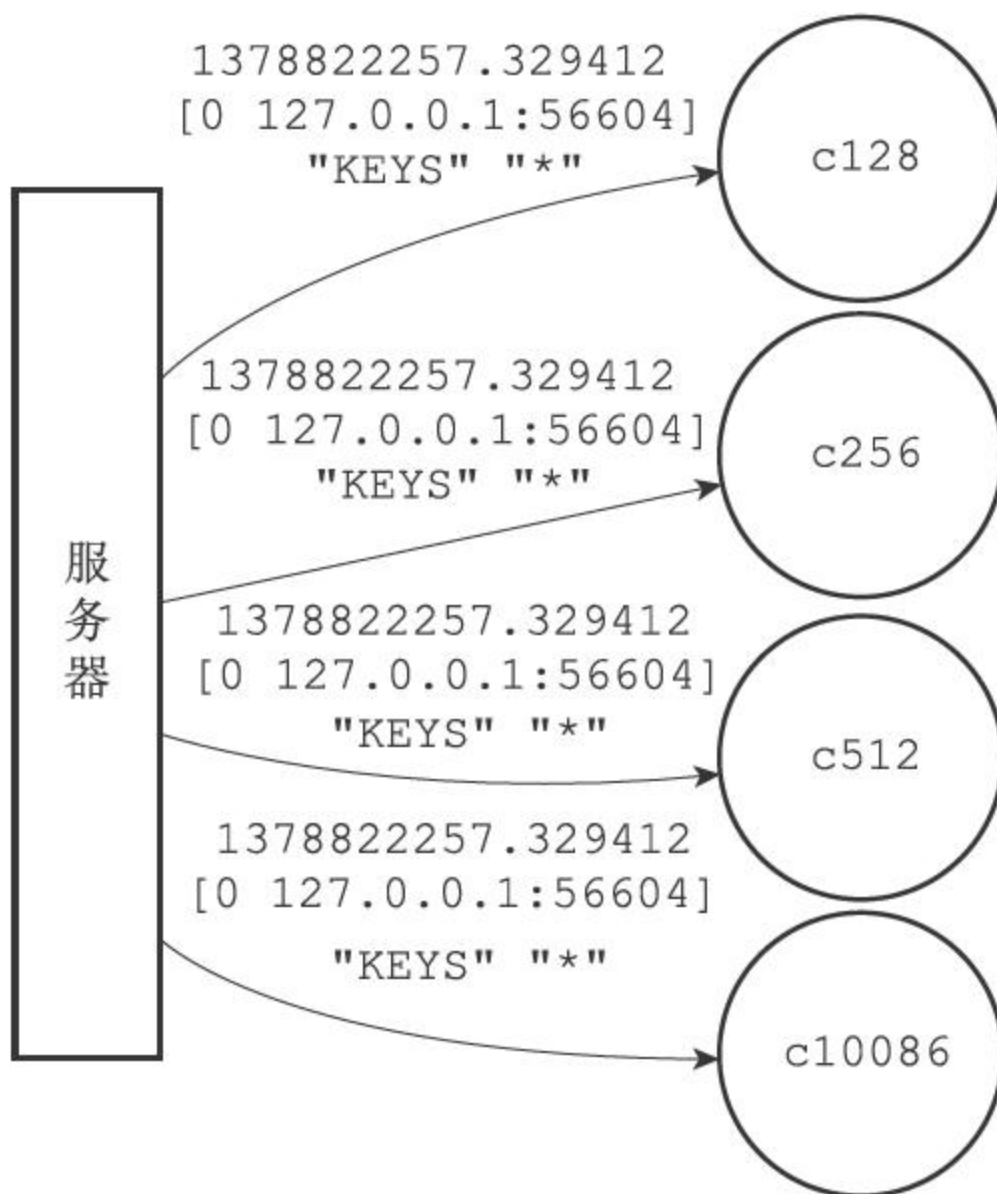


图24-4 消息广播

## 24.3 配置

- 在配置文件中添加 `MONITOR` 命令，用于实时监控 Redis 服务器的运行状态。

- 在配置文件中添加 `REDIS_MONITOR` 命令，用于实时监控 Redis 服务器的运行状态。

- 在配置文件中添加 `monitors` 命令，用于实时监控 Redis 服务器的运行状态。

- 在配置文件中添加 `monitors` 命令，用于实时监控 Redis 服务器的运行状态。